Master of Computer Application

(Open and Distance Learning Mode)

Semester – I

Computer Organization and Architecture

Centre for Distance and Online Education (CDOE)

DEVI AHILYA VISHWAVIDYALAYA, INDORE

"A+" Grade Accredited by NAAC

IET Campus, Khandwa Road, Indore - 452001

www.cdoedavv.ac.in

www.dde.dauniv.ac.in

Program Coordinator

Dr. Anand More
School of Computer Science and IT
Devi Ahilya Vishwavidyalaya, Indore – 452001

Content Design Committee

Dr. Pratosh Bansal
Centre for Distance and Online Education
Devi Ahilya Vishwavidyalaya, Indore – 452001

Dr. C.P. Patidar
Institute of Engineering & Technology
Devi Ahilya Vishwavidyalaya, Indore – 452001

Dr. Shaligram Prajapat
International Institute of Professional Studies
Devi Ahilya Vishwavidyalaya, Indore – 452001

Language Editors

Dr. Arti Sharan
Institute of Engineering & Technology
Devi Ahilya Vishwavidyalaya, Indore – 452001

Dr. Ruchi Singh
Institute of Engineering & Technology
Devi Ahilya Vishwavidyalaya, Indore – 452001

SLM Author(s)

Mr. Mohit Verma
M.C.A.
SCS, Devi Ahilya Vishwavidyalaya, Indore – 452001

Mr. Ashish Panchal
B.E., M.E.
IET, Devi Ahilya Vishwavidyalaya, Indore – 452001

Copyright : Centre for Distance and Online Education (CDOE), Devi Ahilya Vishwavidyalaya

Edition : 2022 (Restricted Circulation)
Published by : Centre for Distance and Online Education (CDOE), Devi Ahilya Vishwavidyalaya
Printed at : University Press, Devi Ahilya Vishwavidyalaya, Indore – 452001

Computer Organization & Architecture

Table of Contents

Introduction

MODULE: I INTRODUCTION

Unit 1 – Introduction

1.0 Introduction

1.1 Unit Objectives

1.2 Digital Computers

1.3 Von Neumann Computers

1.4 Basics of Computer Architecture and Organization

1.5 History of Computers

1.6 Operational Concept

1.7 Summary

1.8 Key Terms

1.9 Check Your Progress

Unit 2 – Digital Logic Circuits

2.0 Introduction

2.1 Unit Objectives

2.2 Boolean Algebra

2.2.1 Boolean Operators 2.2.2 Truth Table

2.2.3 Boolean Identities

2.3 Logic Gates

2.3.1 Common Logic Gates 2.3.2 Universal Gates

2.3.3 Combinational Gates

2.4 K- Map Simplification

2.5 Combinational Circuits

2.5.1 Decoder 2.5.2 Multiplexer

2.6 Arithmetic Circuits

2.7 Sequential Circuits

2.7.1 Basic Latch 2.7.2 Flip- Flop
2.8 Registers and Counters
2.9 Summary
2.10 Key Terms
2.11 Check Your Progress

MODULE: II – COMPUTER ARITHMETIC AND MICROOPERATIONS

Unit 3 – Computer Arithmetic

3.0 Introduction
3.1 Unit Objectives
3.2 Data Representation
3.2.1 Conversion Techniques
3.3 Addition and Subtraction of Binary Numbers
3.3.1 Two's Complement Method
3.4 Multiplication of Binary Numbers
3.4.1 Booth's Algorithm
3.5 Division of Binary Numbers
3.6 Floating-Point Number Representation
3.7 Floating-Point Arithmetic and Unit Operations
3.7.1 Floating-point Addition and Subtraction
3.7.2 Floating-point Multiplication
3.7.3 Floating-point Division
3.8 Binary Codes and Error Detection Codes
3.9 Summary
3.10 Key Terms
3.11 Check Your Progress

Unit 4 – Register Transfer and Microoperations

4.0 Introduction
4.1 Unit Objectives

4.2 Register Transfer Language
4.3 Register Transfer
4.4 Bus and Memory Transfers
4.5 Bus and Memory Transfers
4.6 Logic microoperations
4.7 Shift microoperations
4.8 Arithmetic Logic Shift Unit
4.9 Summary
4.10 Key Terms
4.11 Check Your Progress

Module: III – BASIC COMPUTER ORGANIZATION, DESIGN AND PROGRAMMING

Unit 5 – Basic Computer Organization and Design

5.0 Introduction
5.1 Unit Objectives
5.2 Instruction Codes
5.3 Register Sets
5.4 Instruction Sets
5.5 Machine Cycle, Timings and Control
5.6 Input- Output and Interrupts
5.7 Basic Computer Design
5.8 Design of Accumulator Logic
5.9 Summary
5.10 Key Terms
5.11 Check Your Progress

Unit 6 – Programming the Basic Computer

6.0 Introduction
6.1 Unit Objectives

6.2 High Level, Assembly, and Machine Language
6.2.1 High Level Language
6.2.2 Assembly Language
6.2.3 Machine Language
6.3 Assembler
6.4 Programming Arithmetic & Logic Operations
6.5 Subroutines
6.6 Input- Output Programming
6.7 Summary
6.8 Key Terms
6.9 Check Your Progress

Module: IV – CENTRAL PROCESSING UNIT AND MEMORY ORGANIZATION

Unit 7 – Central Processing Unit

7.0 Introduction
7.1 Unit Objectives
7.2 General Register Organization
7.3 Stack Organization
7.4 Instruction Formats
7.5 Addressing Modes
7.6 Types of Instructions
7.7 Reduced Instruction Set Computer (RISC)
7.8 Summary
7.9 Key Terms
7.10 Check Your Progress

Unit 8 – Memory Organization

8.0 Introduction
8.1 Unit Objectives
8.2 Memory Classification
8.2.1 Read Only Memory (ROM)
8.2.2 Read/ Write Memory (RAM)

8.3 Memory Characteristics and Hierarchy
8.3.1 Cache Memory
8.3.2 Main Memory
8.3.3 Secondary Memory
8.3.4 Virtual Memory
8.4 Memory Management Hardware
8.5 Memory Decoding
8.6 Summary
8.7 Key Terms
8.8 Check Your Progress

Unit 9 – Control Unit
9.0 Introduction
9.1 Unit Objectives
9.2 Control Memory
9.3 Hardwired Control and Micro Programmed Control Unit
9.3.1 Micro Programmed Control
9.4 Address Sequencing
9.4.1 Conditional Branching
9.4.2 Instruction Mapping
9.4.3 Subroutines
9.5 Microprogram Sequencing
9.5.1 Micro Instruction Format
9.5.2 Symbolic Micro Instructions
9.6 Summary
9.7 Key Terms
9.8 Check Your Progress

Module: V- INPUT/ OUTPUT ORGANIZATION, PARALLEL PROCESSING AND MULTIPROCESSORS

Unit 10 – Input/ Output Organization
10.0 Introduction
10.1 Unit Objectives
10.2 Basic Input/ Output Structure of Computers

10.3 Synchronous and Asynchronous Data Transfer
10.3.1 Strobe Control
10.3.2 Handshaking
10.4 Serial and Parallel Communication
10.5 Modes of Transfer
10.5.1 Programmed I/O (Polling)
10.5.2 Interrupt Driven I/O
10.5.3 Direct Memory Access (DMA)
10.6 Priority Interrupt
10.6.1 Daisy- Chain Priority
10.6.2 Parallel Priority Interrupt
10.6.3 Priority Encoder
10.7 Device Drivers
10.8 Standard I/O Interfaces (Buses)
10.9 Bus Arbitration
10.10 I/O Processor
10.11 Summary
10.12 Key Terms
10.13 Check Your Progress

Unit 11 – Parallel Processing
11.0 Introduction
11.1 Unit Objectives
11.2 Parallel Processing
11.3 Pipelining
11.4 Data Dependency
11.5 Handling of Branch Instructions
11.6 Vector Processing
11.7 Array Processors
11.8 Summary
11.9 Key Terms
11.10 Check Your Progress

Unit 12 – Multiprocessors

12.0 Introduction

12.1 Unit Objectives

12.2 Characteristics of Multiprocessors

12.3 Types of Multiprocessors

12.4 Interconnection Structures

12.4.1 Time-Shared Common Bus

12.4.2 Multiport Memory

12.4.3 Crossbar switch

12.4.4 Multistage switching network

12.4.5 Hypercube system

12.5 Interprocessor Arbitration

12.5.1 Serial Arbitration Procedure

12.5.2 Parallel Arbitration Logic

12.5.3 Dynamic Arbitration Algorithms

12.6 Inter-Processor Communication And Synchronization

12.7 Symmetric Multiprocessors

12.8 Summary

12.9 Key Terms

12.10 Check Your Progress

INTRODUCTION

Today technology has made its roots in the day to day life of humans. We use computers in every way possible or rather we have become completely dependent on computers by now. A normal being is just concerned about the work the computer does for him but unaware of the internal operations, functions, and programs of the computer system. It is essential to have an insight into how the computer system works. Computer Organization and Architecture deal with the study of internal working, structuring distinct functional modules, and implementation of a computer system. Computer Architecture means to design the basic structure of a computer system while computer organization is concerned with the practical implementation of this carefully designed computer architecture in terms of hardware attributes.

This study material reveals the basic internal structure and functioning of a modern-day computer system. The evolution of technology from vacuum tubes based computers to supercomputers, historical background, and operational concept of computers, Von Neumann's structure of computers are discussed in Module-1 of this course. It also explains the fundamentals of the digital logic circuits, Boolean algebra, map simplification, different logic gates, and different logical circuits. This helps in gaining knowledge about the basic circuits like combinational and sequential circuits and logics involved in designing a computer system.

Unit- 3 and 4 of Module: II gives an extended outlook on the basic computer arithmetic operations including different techniques to perform addition, subtraction, multiplication, and division of binary numbers (signed and unsigned). The module also depicts the basic features of a Register transfer language and different microoperations.

The next module, i.e. Module: III illustrates the basics of the computer processor, its architecture, and organization. It explains the concept of instructions, interrupts, register sets, stack organization, and machine cycles to make the readers understand how the internal circuitry of a computer

works. Unit-6 of this module follows the concept of programming in the computer system. It focuses on the concept of High- level, machine, and assembly language programming, subroutines, Input-output programming of the processor.

In Unit- 7 and 8 of Module: IV, the different Addressing modes and discussion about reduced instruction set computing (RISC) and complex instruction set computing (CISC) types of processors are elucidated. The discussion follows the memory system of a computer in detail. The memory classification including ROM, RAM, cache memory, and virtual memory are featured in the module.

Module: V illustrates the detailed discussion about the Input-Output Unit and the internal structure of the control unit of the computer. The fundamental characteristics of data transfer through Programmed I/O, Interrupt driven I/O, and Direct Memory Access (DMA) are explained in the module. It also elaborates on the basic characteristics of multiprocessors. Parallel Processing and Multiprocessors are also discussed in Unit-10 and 11 respectively.

This content is designed comprehensively and follows a simple approach, keeping in mind the syllabus of the program. It exhilarates interest and is sure to stimulate knowledge among the readers. The purpose is to acquaint the readers with the principle and design of computer organization and architecture. Numerous figures and tables, key terms help in simplifying learning about the subject. The 'Check Your Progress' section intends the readers to test their knowledge. It is hoped that the language and the content demonstration is coherent to the readers and will enhance their learning in the best way possible.

MODULE: I

INTRODUCTION

Unit 1 – Introduction

Structure

1.0 Introduction

1.1 Unit Objectives

1.2 Digital Computers

1.3 Von Neumann Computers

1.4 Basics of Computer Architecture and Organization

1.5 History of Computers

1.6 Operational Concept

1.7 Summary

1.8 Key Terms

1.9 Check Your Progress

1.0 Introduction

With the advancement in science and technology, the computer has become an essential need for humans. We are totally dependent on computers for the day to day work in different ways. There is hardly any field which is not affected by computers. The basic and theoretical definition of a computer is not unfamiliar with this world. It is stated that a computer is an electronic machine that performs different operations as per the need of the user. The raw data as an input is provided to the computer through input devices, then the processor processes this data by performing various operations, and finally, the output is obtained from the output devices by the user. However, it is strange that even among some present-day people; there is minimum awareness about the overall working mechanism of a computer, although these people may be daily using a computer in any manner like laptops, mobiles, notepads, etc. They continue with a mindset that the detailed knowledge of computer architecture and structure is confined to computer engineers and others related to the information technology stream, but it is not so. The basic working mechanism is studied under Computer Organization and Architecture. Figure 1.1 shows

the basic computer architecture that comprises Input and Output devices; Central Processing Unit (CPU) that is subdivided into Control Unit (CU) and Arithmetic Logic Unit (ALU); Main memory and external storage. Each block contributes to different functions of the system, which we will discuss in the following units.

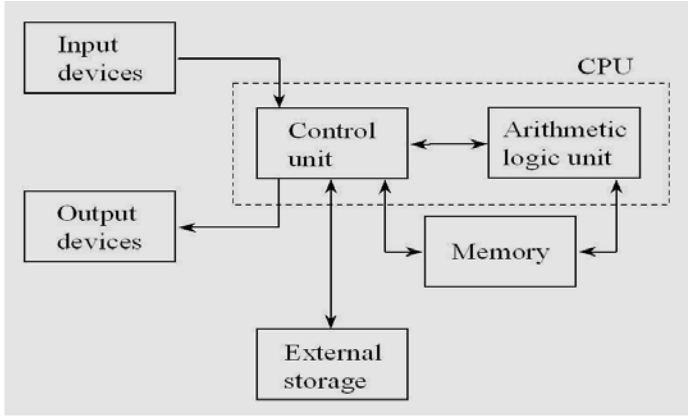


Figure 1.1 Basic Computer Architecture

1.1 Unit Objectives

On completion of this unit, one will be able to:

- Gain knowledge about the basics of computers and its fundamentals.
- Discuss the brief history of the evolution of computers.
- Learn the general operational concepts of a computer.

1.2 Digital Computers

Digital computers refer to the computer system that operates on the concept of discrete numbers. They are generally used for numerical applications of the computer system. Digital computers work on the binary number system, having two states 0 and 1. The smallest unit of the binary number system is a *bit.* Group of bits represents any form of information in the system. Different coding techniques are used to perform various operations in digital computers. A complete set of instructions can be represented using a group of bits to perform any type of operation.

A basic computer functions using two entities: software and hardware. The

hardware part includes all the peripheral devices (input and output) connected to the computer and have physical appearance for the users. On the other hand, the software refers to the internal programs and instructions that are processed by the computer. A computer program is a combination of a set of instructions to be processed by the computer. All the computer programs require a basic platform for proper functioning and this interface between the computer and the user is known as the operating system.

Computer architecture and organization are two slightly different aspects in terms of definitions. Computer architecture generally includes design-related issues while computer organization deals with the implementation of the architectural modules. Computer designing and implementation is generally related to the hardware designing of the system. It takes an account of what and how hardware should be connected in a system. Computer architecture is basically concerned with the internal structure of the computer including the format of information, instruction sets, memory management, etc. It can be observed that during the passing years, there was not much change in the architecture of a computer but its organization has sustained various changes and is still continuing. We will discuss the history or evolution of computers, for a better understanding of these changes in the organization.

1.3 Von Neumann Computers

The early day's computers were bound to process one instruction at a time and most of the time the instructions were loaded in the system manually by the operators. It was time taking and less effective. Due to the absence of any provision to store the instructions to be executed later, it was more difficult. The function of such computers was very specific and fixed. It was not possible to program them regularly, for example, calculators.

To overcome this issue, in 1945, John Von Neumann Architecture came to the rescue. According to the Von Neumann architecture, one can encode and even store the instructions in the memory of the computer so that they can be executed whenever required. While executing any program, the desired

instruction is fetched from the memory and then decoded to generate the required output. Figure 1.2 shows the basic structure of Von Neumann architecture. As depicted from the figure, it can be observed that the control unit is being considered as a part of the processor only. The control unit generates control signals that are responsible for the proper functioning of the instructions of the computer system. The fetching, decoding, and execution of instructions from the memory reduce the execution time and impart faster speed to the computer system. This cycle of instructions in the Neumann architecture is termed as ***instruction cycle.***

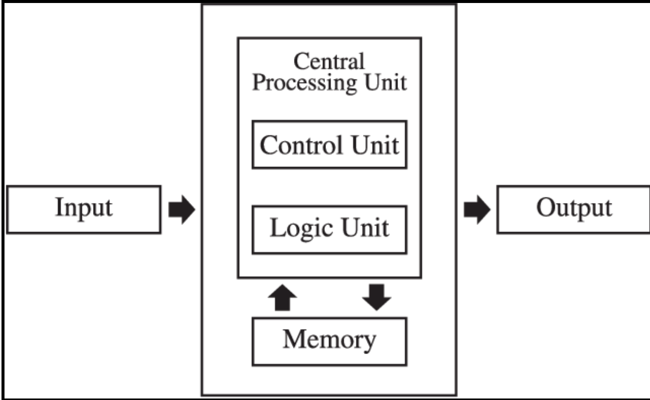


Figure 1.2 Basic structure of Von Neumann Architecture of Computer

Von Neumann architecture proved to be a revolution since its introduction. It formed the basis for computer designing for several decades. The only issue with this architecture was that only one instruction was being fetched at a time, or the memory could be accessed only once a while. As this architecture was deprived of sequential instruction execution, this state was referred to as *'Von Neumann Bottleneck'.* The data and instructions both are stored in the same memory and use the same bus for data transfer. This can result in low-performance computations.

To overcome the above mentioned drawback of Von Neumann architecture, ***Harvard Architecture*** was considered. Harvard architecture comprises separate storage and signal path for data and instruction. This results in simultaneous instruction execution and data processing.

1.4 Basics of Computer Architecture and Organization

We all are aware of the external structure of a personal computer or desktop that contains input devices such as a keyboard or a mouse; output devices like monitor display screen or a printer; and a CPU. Different types of computers can be seen at different places depending upon their need, for example, a workstation can be seen at some designer's office where the basic computer is attached to other peripherals like scanner and digitizer. Here, we will discuss an overview of the computer hardware briefly, used in the basic computer architecture shown in Figure 1.1. Let us first discuss the basic working of a computer.

- **Input Devices:** Like any other machine, it is required by the computer to get the prior instructions and data from the user to process the information. The input devices are used to feed instructions, data, and programming to instruct the computer to perform the required function on the input data. The commonly used input devices are keyboard, mouse, touchpad, scanner, microphone, disks, etc.
- **Processing:** The process of performing operations and functions on the raw data provided by the user is known as the processing of data. The CPU performs all the required operations and calculations with the help of ALU. The control unit (CU) controls all these operations. The result of the processed data is then provided to the user via output devices and also stored in the primary storage of the computer.
- **Storage:** It is essential for the computer to store the data which it has to process and which it has already processed. Some data is to be stored permanently while some are meant to be stored temporarily. On this basis, the storage unit is subdivided into two types: **Primary Storage and Secondary Storage**. Primary storage is the internal memory of the computer that is temporary and small in size while secondary storage has large storage capacity and they store the data permanently. Figure 1.3 shows the basic classification of the memory of the computer. Primary storage includes RAM (Random Access Memory) and ROM (Read

Only Memory). RAM is volatile memory i.e. it gets erased when the power is switched off while ROM is a non-volatile memory i.e. it stores the data permanently and retains the data even if the power gets off. The CPU can only read the contents of ROM and cannot edit it while the content of RAM can be read and written both by the CPU whenever required. The primary memory also includes the Cache memory and Registers which are used to store the data internally and are temporary in nature.

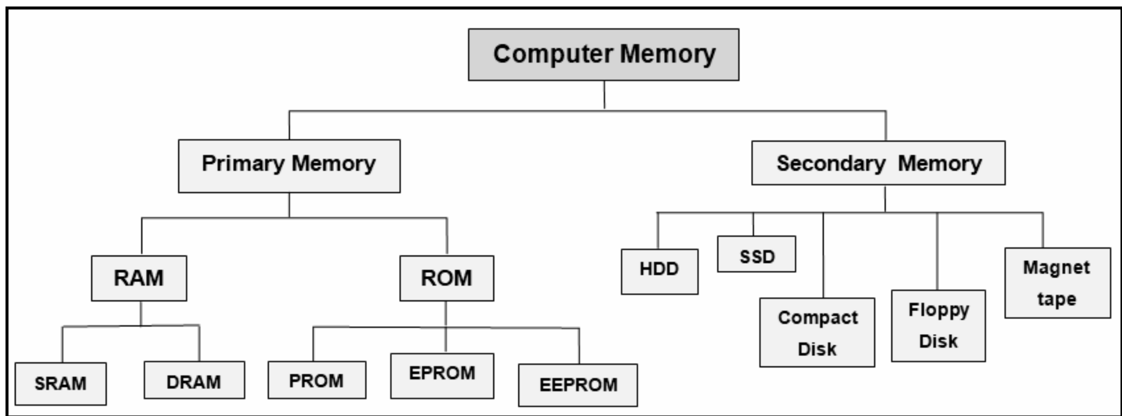


Figure 1.3 Classification of Computer Memory

The secondary storage devices include Hard disk drives (HDD), memory cards, compact disks (CD), floppy disks, and magnetic tapes.

- **Output Devices:** The output devices are used to display the result of the processed data to the user. The result can be in an audio, visual, or printed form. The most common output devices are monitor display and printer. Other output devices are a projector, speakers, headphones, etc.
- **Power Supply:** The power supplied to the computer depends upon its type and size. Earlier computers used more power and slowly with the decrease in size, the power consumption of the computer also decreased. The Power Supply Unit (PSU) is the internal power unit of the computer that modulates the main AC supply to a low-voltage DC power supply. Switched Mode Power Supply (SMPS) is the popular power supply used by modern computers.

- **Motherboard:** It is the main circuit board of the computer in the form of a printed circuit board. It comprises a microprocessor and other necessary components that are interconnected with the help of cables. The motherboard provides an interface between the input/ output devices and the storage of the computer.

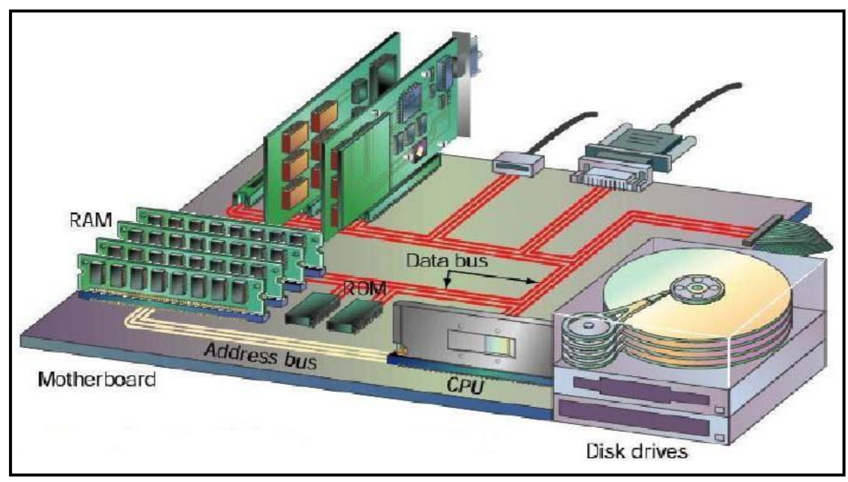


Figure 1.4 Motherboard and other functional units of a computer

(**Source-** https://www.slideshare.net/mobile/AmosNdubi/how-computers-transform-data-to-information-lesson-3-bso)

1.5 History of Computers

The present scenario of computers is more accessible, accurate, user- friendly, speedy, simple, compact, and easy to use. However, the earlier picture was much different. The earlier generations of computers were complex and time taking. The foremost step towards automatic computing was taken by the well-known European scientist, **Pascal,** long back in 1642 when he fabricated a mechanical calculating machine, which was capable of only addition and subtraction. Later, it was improvised by a German mathematician, **Leibniz** and the features of multiplication and division were also added to it. Computers came into existence about 150 years later, in the early nineteenth century. A mathematics professor, **Charles Babbage** invented the first generation of computers and then the technology made its way. The growth of computers can be divided into five phases, termed as Generations of Computers. Each

generation is superior from its predecessor in terms of processing, capabilities, appearance, size, switching circuits, and the technologies used. They are:

- First Generation Computers (1940-1956)
- Second Generation Computers (1956-1963)
- Third Generation Computers (1964-1971)
- Fourth Generation Computers (1971-Present)
- Fifth Generation Computers (Present and Ahead)

Figure 1.2 Five Generations of Computer

(Source- https://www.inhindis.com/generation-of-computer/)

Let's discuss the five generations of computers in detail.

1. **The First Generation Computers- Vacuum Tubes (1940-1956)**

   The first era of computers initiated with an introduction to Vacuum tubes. A vacuum tube is a glass tube filled with certain filaments to generate electricity. It was used to control and increase the intensity of the electronic signals. They served for calculations, storage, and control of the machine. Magnetic drums were used for storage of data, while input was provided through punch cards. The use of machine languages was prominent for programming.

One of the pioneers of this generation was the Electronic Numerical Integrator and Computer **(ENIAC)** which was developed at the University of Pennsylvania by **J. Presper Eckert and John V. Mauchly**. It was very large in size, heavy in weight and high in power consumption, approximately, 30-50 feet long and covered the floor area of about 150m^2, with the weight of 30 tons, the power consumption of nearly 140kW and consisted of more than 18,000 vacuum tubes, 10,000 capacitors, and about 70,000 registers. Interestingly, ENIAC used Decimal representation unlike binary representation of numbers. For example, to represent a digit of a decimal integer, there was a group of 10 vacuum tubes, each signifying a decimal number between zero to nine. Due to enlarged size, large area requirement, a large amount of heat emission, made these computers complex to use. However, they created awareness for the development and further usage of computers.

Another marked contribution was made by **Von Neumann** by designing the Electronic Discrete Variable Automatic Computer (EDVAC) which incorporated memory for both data and the pre-stored programs. Due to rapid access to data and instructions, the operation was much faster and the computer could make logical decisions internally. Later, **Eckert and Mauchly** introduced the first commercially successful computer of this era, the Universal Automatic Computer (UNIVAC), in the year 1952.

2. Second Generation Computers- Transistors (1956-1963)

The second generation of computers was based on transistors during the year 1956 to 1963. Transistors were invented in Bell Laboratory in the year 1947 and were used as a replacement for vacuum tubes due to their smaller size and increased efficiency. A Transistor is a device that is composed of semiconductor material that amplifies the signal and can act as a switch (ON and OFF). Along with transistors, other solid-state components like diodes and magnetic core memories instead of magnetic drums for storage of data were used. High-level programming was introduced which used mnemonic codes and symbols. Some of the high-

level languages were FORTRAN (1956), ALGOL (1958), and COBOL (1959).

Digital Equipment Corporation **(DEC)** developed a minicomputer, Programmed Data Processor-1 **(PDP-1)** which was said to be a milestone of this era. It had a display screen of 512 x 512 pixels, a memory of 4096 locations, each location of size 18 bits, speed of 2,00,000 instructions per second. Later, DEC continued with the series with **PDP-8** as an improvised version. Meanwhile, **IBM** also contributed to this generation by the invention of the series of computers, named 700 and 7000. It was observed that the **7094** version of the series by IBM was a 36-bit machine with a memory of 32,536, dominated the market at that time. Apart from this, Control Data Corporation **(CDC)** along with **Cray** encountered more superb inventions **6600** and **7600** of this generation.

3. **Third Generation Computers- Integrated Circuits (1964-1971)**

With the advancement in devices made of transistors, technology took a rapid growth as minimizing the size of transistors and assembling them on a single chip for remarkably increasing the speed and efficiency of computers. A number of transistors along with other components like registers, capacitors, switching devices are assembled on a single chip made of silicon, known as Integrated Circuits (IC). The process of developing the ICs started with small scale integration (**SSI**). The arrangement tends towards incorporating more components on a single chip, thus changing the process to large scale integration (LSI) and later very large scale integration (VLSI). The invention of ICs proved to be a landmark in the development of computers and other electronic devices of the third generation of computers.

It was first observed by **Gordon Moore** in the year 1965, that every year the number of components (transistors, diodes, etc.) will get double on the single-chip or wafer (silicon base for manufacturing ICs), however, it is noted that the number moderately doubles in approximately every 18 months now. It was predicted that this trend will be followed at least for

a decade, while it is interesting to note that it is still being followed. This is famously known to be **Moore's Law.** In accordance with Moore's law, the miniaturization or shrinking of the size of the components are some highlighted facts for the growth of technology in ICs.

Silicon is chosen to be an elementary and ideal material for designing semiconductor-based devices due to its distinctive properties and ability to be doped with other specific materials (Boron, Arsenic, and Phosphorus) to alter its properties. The standard size of an IC is considered to be less than 0.25 square inches that may have millions of devices integrated on it. The basic mechanics of a computer consists of several ICs mounted on printed circuit boards (PCB) that persist in different functions. There are different ICs for different processes, for example, an IC for memory functioning, another IC for processing of information, etc. Since large circuits were integrated on a single chip, another name given to IC technology was "microelectronics". Due to their small size, the cost of the computers of this generation became low with high processing speed and large memory space. The fast solid-state memories substituted the core memory of previous-generation computers. High- level languages like BASIC (Beginners All-purpose Symbolic Instruction Code) were utilized in this era. The mainframe computer **IBM System/ 360** and the minicomputer **DEC PDP-8** were the important inventions of this generation.

4. **Fourth Generation Computers (1971- Present)**

With the advancement in the fabrication of Integrated Circuits through **VLSI** (Very Large Scale Integration- about 10,000 components per chip) on a single chip, the fourth generation of computers was characterized by the use of Microprocessors, resulting in increased data processing capacity. A microprocessor is an integrated circuit that comprises all the distinct functions of the Central Processing Unit (CPU) of the computer. It is capable of accumulating processing for all arithmetic and logical functions of the computer. For example, the circuit of multiplication of

two numbers can be added to the same chip which is used for switching ON/OFF of the computer. One microprocessor may contain hundreds of integrated circuits or more. When the CPU of the computer is placed on a single chip then the computer is termed as a microcomputer.

This era of computers marked the growth in Microelectronic circuits and digital electronic circuits in different fields. For external storage, floppy disks and magnetic tapes were in use while semiconductor memory chips were considered for main memory. Operating systems like **MS-DOS** and **MS-Windows** were introduced in this generation. Also, the beginning of computer networking through **LAN** and **WAN** was seen by this generation of computers. Another important contribution to this generation was the Graphical User Interface (**GUI**) which developed visual graphics for computer software for the users to make it more attractive and easy to use. It was the time of workstations, personal computers (PC), and microcomputers, which were advantageous in terms of compactness, high speed, low price, user-friendly, fast data processing, less power consumption, and high storage capacity.

Intel 4004 was the first microprocessor chip being developed in this era that included CPU, input/ output control, and memory on a single chip. Other important innovations included **Apple II, TRS- Radio Shack, and BBC MICRO.** In 1981, IBM collaborated with Intel for an important innovation, **IBM PC with Intel 8088** microprocessor. Later, the series of more powerful microprocessors like **80186, 286, 386, 486, Pentium series,** and **Core 2 Duo** processors invaded the market of computers.

5. Fifth Generation Computers (Present and Ahead)

The most recent generation of computers is based on Artificial Intelligence and Neural Networks and is still under development. The main aim of the fifth generation of computers is to develop such computers which are self-accessible, self-organizing, learning, and responding to natural languages, ultimately the computers should behave like humans. The voice recognition feature utilized by various

recognized companies like Amazon and Google to develop such devices that can work on just verbal instructions is an example of artificial intelligence technology. Other examples can be seen in online games, education, and intellectual modules for kids and many more fields. However, at present, there is no such computer developed that exhibits artificial intelligence completely but it will soon get into the race. The enhancement in technology has resulted in extremely high speed in the computers. The quantum computers have gained a lot of focus for the future research of technology and developing supercomputers. The technology used for designing such computers will be based on **ULSI** (Ultra Large Scale Integration) and Nanotechnology which is framing a new world of research for miniaturization of the size of the components on a single chip or making way for Nanoelectronics.

Table 1.1 Comparison of five generations of computers

Generation / Criteria	Time Period	Technology	Size	Language	Speed	Storage
First Generation	1940-1956	Vacuum Tubes	Largest	Machine	Slowest	Magnetic Drums
Second Generation	1956-1963	Transistors	Large	Assembly	Slow	Magnetic Core Memories
Third Generation	1964-1971	Integrated Circuits	Medium	High- Level	Medium	Solid State Memories
Fourth Generation	1971-Present	Microprocessors	Smaller	High- Level	Faster	Semiconductor Memory Chips
Fifth Generation	Present and Ahead	Artificial Intelligence	Smallest	High- Level	Fastest	Magnetic RAM or spintronics based devices, etc.

1.6 Operational Concept

The basic need for performing any task on the computer is that there should be a proper list of an instruction set or programming stored in the primary memory. The user makes use of high-level language for writing the instructions

or programs which are termed as source codes. The compiler then translates the high- level source code into machine language code that is stored in the memory. The CPU fetches these instructions from the memory and executes them and then again sends the processed data to be stored in the memory. If there are many instructions that are to be executed concurrently, then the CPU works in a multiplexed form. The basic operational concept explains how the execution of the instructions is carried out by the CPU. This includes several parameters and steps that are being discussed one by one.

- **Processor Clock:** The clock is the main parameter of any process that depicts the proper timing of the process. It is a digital clock that produces ON and OFF states at regular time intervals. Figure 1.5 shows the oscillating cycle representing the range from 0-5 volts with a cycle time of one pulse. Here, we can see two different edges of the clock signal i.e. rising clock edge and falling clock edge, also known as positive edge and negative edge respectively, displaying the transition of the clock for increment or decrement of the program counter. For reference, if the clock speed of any processor in any electronic device is given as 4 GHz, then the cycle time for one process can be calculated by 1/ clock speed, which shows how fast the processor can execute one operation. To execute any instruction, the process of operation is divided into a sequence of steps that can be completed in one clock cycle each. Today, such processors are available in the market having the speed range of a hundred million cycles to over a billion cycles per second.

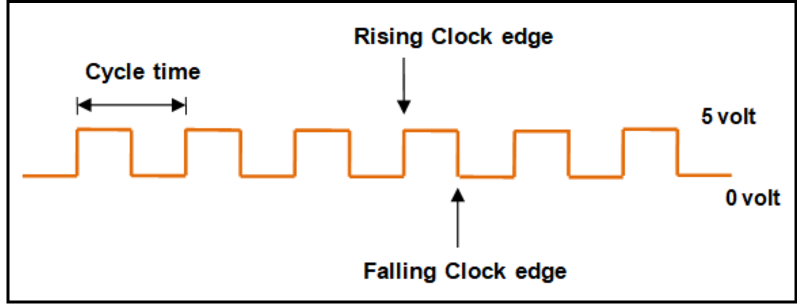


Figure 1.5 Processor Clock cycle

- **Program Counter (PC):** The processor comprises a binary/ digital counter called program counter that gets incremented by one on every cycle of the clock signal. It is a register that consists of a group of bits and contains the address of the next instruction to be accomplished from the memory. A register can be considered to be the smallest storage unit in the processor that can hold an instruction, memory address, or any other bit sequence. The data when entered by the user gets stored in the main memory of the computer then it is fetched by the CPU, so the program counter keeps the count of these instructions and gets updated when the execution of one instruction is over.

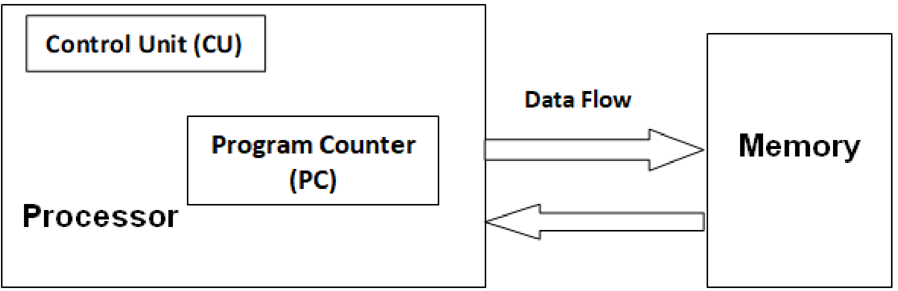


Figure 1.6 Reading Data from Memory

- **Instruction Fetch:** The process of fetching the data to be executed by the CPU in the form of instruction is known as instruction fetch. As soon as the instruction is fetched, the information of this instruction byte gets available in the processor. With the next clock cycle, the program counter is incremented by one and waits for the next instruction fetch process to be completed.
- **Instruction Decode:** As soon as the processor reads the instruction byte of the fetched instruction, it starts understanding it; this process is called Instruction Decoding. An increased number of instructions makes the decoding process complex. There are certain techniques to reduce this complexity which we will discuss in upcoming units.
- **Instruction Execute:** After the decoding of the instruction, the execution of the instruction is started by the processor immediately.

Thus, the process of incrementing the PC, fetching, decoding, and executing the instruction continues. In earlier processors, all these steps occupied different time slots and resulted in the slow speed of the processor, but with modern processors, the issue was solved by accomplishing the whole process simultaneously. This is called Pipelining.

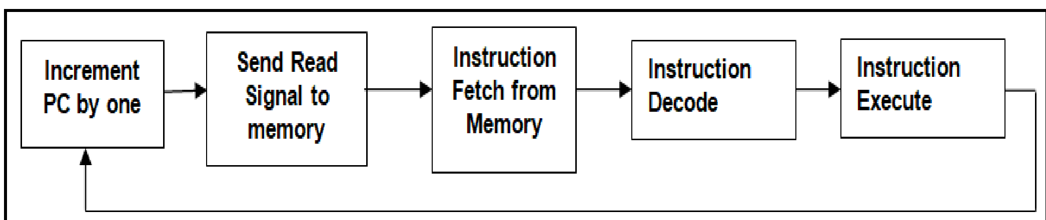


Figure 1.7 Basic Operational Flow of a processor

Besides the Program counter (PC), there are several other registers in the processor that are used for different purposes. Some are used to establish the connection between the memory and the processor while some are used for fetching instructions from the memory for execution. They are:

- **Instruction Register (IR):** It is a register that contains the instruction that is presently under execution.
- **General Purpose Registers:** There are n- general-purpose registers ranging from $\mathbf{R_0}$ to $\mathbf{R_{n-1}}$ which are used to store data generally when required.
- **Memory Address Register (MAR):** It holds the address of the location in the memory that is to be accessed.
- **Memory Data Register (MDR):** It includes the data that is to be read or written into, out of the memory address.

The complete steps for operation flow are:

1. Initially, the set of instructions called a program is either in the memory itself or usually received through Input from the user.
2. The Program Counter (PC) points at the first instruction of the program and the execution starts.

3. The instruction from the PC is transferred to **MAR** and it sends a ‘Read’ signal to the memory as shown in figure 1.7.
4. When the memory access time gets over, the address is read out of the memory and transferred into **MDR**.
5. The contents of MDR are now loaded into **IR** and the instruction gets ready to be decoded and executed. After the execution of an instruction, the address of the location where the result is stored is sent to MAR.
6. Then the PC gets incremented by one to indicate that it is ready for the next instruction to be executed.

1.7 Summary

- According to the Von Neumann architecture, one can encode and even store the instructions in the memory of the computer so that they can be executed whenever required. The only issue with this architecture was that only one instruction was being fetched at a time, or the memory could be accessed only once a while.
- The evolution of the computer system in five generations depicts the enhancement in quality, efficiency, accuracy, storage capacity, and speed of computers.
- The building blocks of a computer include both hardware and software components. The motherboard or processor is the heart of the computer. The operating system is known as the interface between a user and computer, different programs and languages form the software building blocks of the system.
- The program counter is a register that contains the information of the next instruction to be executed. Besides PC, instruction register (IR), Memory Address Register (MAR), Memory Data Register (MDR), and general-purpose registers are other registers that are used in the basic operational concept of computers.
- The program counter is a register that contains the information of the next instruction to be executed. Besides PC, instruction register (IR),

Memory Address Register (MAR), Memory Data Register (MDR), and general-purpose registers are other registers that are used in the basic operational concept of computers.

1.8 Key Terms

- **SRAM:** Static Random Access Memory is a semiconductor that holds the data in a static manner and does not change rapidly but it is volatile in nature.
- **DRAM:** Dynamic Random Access Memory is a type of semiconductor memory that stores each bit of data in a separate capacitor. The storage is dynamic i.e. the content can be changed whenever required. It can store more data than SRAM but requires more power.
- **PROM:** It is a programmable read-only memory that can be programmed once by the user according to the need and the data remains permanent in PROM. It is a non-volatile memory.
- **EPROM:** Erasable programmable read-only memory is a type of ROM that can be erased and reused, unlike PROM. The memory is erased using UV-rays. The EPROM chip has to be removed from the system and then erased and reprogrammed.
- **EEPROM:** Electrically erasable programmable read-only memory can be erased and reprogrammed repeatedly by applying a higher voltage pulse. There is no need to remove the chip each time, it is user-modifiable. It is also termed as an upgraded version of EPROM.

1.9 Check Your Progress

Q1) Explain the purpose of the Program Counter.

Q2) Write a short note on General-purpose registers in the computer system.

Q3) Discuss the classification of Memory in detail.

Q4) Explain the basic operational function of the computer system.

Q5) Describe the generations of computers in detail.

Q6) What is the advantage of Harvard Architecture?

Q7) Define Von Neumann architecture of computers with its drawbacks.

References:

Computer System Architecture, M. Morris Mano

Computer Architecture and Organization, Subrata Ghoshal, Pearson Publication

http://www.egyankosh.ac.in/bitstream/123456789/10950/1/Unit-1.pdf

https://www.academia.edu/35443462/Computer_Generations

http://www.idconline.com/technical_references/pdfs/information_technology/Basic_Operational_Concepts_of_Computer.pdf

https://www.researchgate.net/publication/336700280_History_of_computer_and_its_generations

https://www.geeksforgeeks.org/computer-organization-von-neumann-architecture/

Unit 2 – Digital Logic Circuits

Structure

2.0 Introduction

2.1 Unit Objectives

2.2 Boolean Algebra

2.2.1 Boolean Operators 2.2.2 Truth Table

2.2.3 Boolean Identities

2.3 Logic Gates

2.3.1 Common Logic Gates 2.3.2 Universal Gates

2.3.3 Combinational Gates

2.4 Map Simplification

2.5 Combinational Circuits

2.5.1 Decoder 2.5.2 Multiplexer

2.6 Arithmetic Circuits

2.7 Sequential Circuits

2.7.1 Basic Latch 2.7.2 Flip- Flop

2.8 Registers and Counters

2.9 Summary

2.10 Key Terms

2.11 Check Your Progress

2.0 Introduction

Digital Logics form the fundamentals of the computer system. The computer works on the binary logic of '0' and '1' bit that is considered as the smallest unit. All the arithmetic and logical operations and others are performed on the basis of binary systems and the digital circuits. The computer accepts the data from the user in a user-friendly language and then the data is converted into machine codes in binary form. After execution, the result is again converted to the language understandable by the user.

Following are the units of a binary system that are used in computers:

- **Bit:** 0 and 1
- **Nibble:** Group of 4 bits.
- **Byte:** Pair of 2 nibbles or 8 bits.
- **Word:** Group of 2 bytes or 16 bits.
- **Double Word:** Combination of 2 words or 32 bits.
- **Quad/ Long Word:** Group of 2 double words or 64 bits and so on.

The basic function of a computer is governed by either sequential or combinational circuits using digital logic. There are two basic modules of data processing and control operation, data storage and data flow control. These modules are implemented by circuits based on digital logic using flip- flops or logic gates that are said to be the fundamentals of digital circuits. The operation is performed in such circuits using the mathematical foundation called Boolean algebra that helps in analyzing and designing the circuits. In this unit, we are going to discuss all these logical circuits in detail.

2.1 Unit Objectives

After completion of this unit, the reader will be able to:

- Understand the basics of Boolean algebra and Logic gates.
- Analyze combinational, arithmetic, and sequential circuits.
- Study the basics of registers, counters, and memory circuits.

2.2 Boolean algebra

Boolean algebra is the mathematical foundation that is used to design the digital circuitry and other digital systems and analyze their behavior and fundamentals. **George Boole**, an English mathematician proposed the basics of Boolean algebra in 1854. Later, in 1938, **Claude Shannon** suggested that this algebra can be used in designing switching circuits in the digital electronics field. The field of computer architecture and organization is concerned with digital electronics due to two major functions to be implemented. They are storing Boolean information, which is implemented

using registers and flip-flops, and transferring this Boolean information from one place to another with the help of logic gates. The simplest analogy of Boolean algebra is a switch that has two states ON and OFF representing logic 1 (True or positive) and logic 0 (False or negative) respectively. Certain variables and operators are used in Boolean algebra that is termed as Boolean operators.

2.2.1 Boolean Operators

The basic logical operations include AND, OR, and NOT, generally represented as '.', '+', and '—' (an over-bar sign on the variable) respectively. Let us assume that A and B are two variables both having two states (0 and 1), then

A AND B = A.B

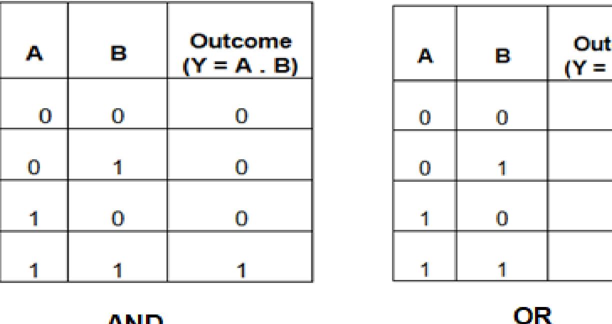
A OR B = A + B

NOT A = $\bar{A}$

- **AND operator:** The operation AND is true only if both the variables A & B are true. That means if we consider the example of a switch, both A & B are two switches. When both A & B are closed (on) or open (off) then only the AND operation will give true outcome. If anyone of them is on and others are off then the outcome will be false.
- **OR operator:** This operation is true if both A & B are in different states. If either or both A & B switches are closed (on) then only the outcome of the operation will be true.
- **NOT operator:** It is the simplest operator as it represents the inverse of the original state. The outcome of NOT for logic 0 (false) is 1 (true) and for logic 1 (true) is 0 (false).

2.2.2 Truth Table

A truth table is a representation of Boolean operators in a tabular form. It becomes easy to understand the Boolean expressions with the help of truth tables. It represents OFF state as 0 and ON state as 1. Figure 2.1 shows the truth table of AND, OR and NOT operators.

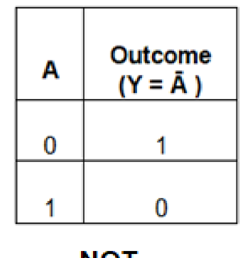


A	B	Outcome (Y = A . B)
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	Outcome (Y = A + B)
0	0	0
0	1	1
1	0	1
1	1	1

OR



A	Outcome (Y = $\bar{A}$)
0	1
1	0

NOT

Figure 2.1 Truth tables of AND, OR, and NOT operators

2.2.3 Boolean Identities

To simplify and solve the different functions of Boolean algebra, there are several identities available. These are some basic rules that are followed in Boolean algebra to solve the Boolean expressions. Table 2.1 shows the different Boolean identities, considering the Boolean operators and variables.

Table 2.1 Fundamental Boolean Identities

Identities (AND form)	Identities (OR form)	Name
1.A = A	0+A = A	Identity Law
0.A = 0	1+A = 1	Null Law
A.$\bar{A}$ = 0	A+$\bar{A}$ = 1	Inverse Law
A.B = B.A	A+B = B+A	Commutative Law
A.A = A	A+A = A	Idempotent Law
A.(B.C) = (A.B).C	A+(B+C) = (A+B)+C	Associative Law
A.(B+C) = (A.B)+(A.C)	A+(B.C) = (A+B).(A+C)	Distributive Law
A.(A+B) = A	A+A.B = A	Absorption Law
$\overline{A.B} = \bar{A}+\bar{B}$	$\overline{A+B} = \bar{A}.\bar{B}$	DeMorgan's Theorem

2.3 Logic Gates

Logic gates are said to be basic components in the field of digital electronics. They are used to generate simple to complex digital circuits. It can have one or more variables as input depending upon the number of signals and generate

one result after performing logical operations between the inputs. These logic gates accept binary inputs 0 (False/ OFF) and 1 (True/ ON) and perform the desired operations on such signals. They are generally designed with the help of electronic switches such as diodes and transistors. With the help of logic gates, Boolean expressions can also be implemented in electronic circuitry.

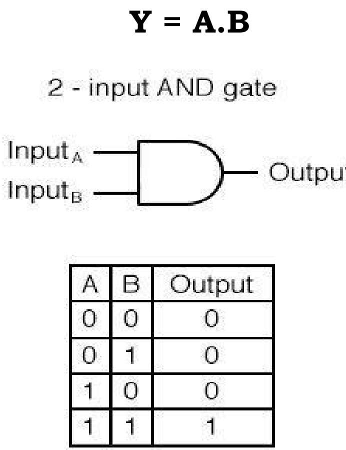
Gates are categorized as **Primary Gates** (AND, OR, and NOT), **Secondary/ Derived Gates** (NAND, NOR- known as Universal gates and XOR, XNOR- known as Combinational gates). The symbol and representation of these logic gates are prescribed by ANSI/ IEEE standards. Let us discuss them in detail.

2.3.1 Common Logic Gates

We will discuss the common or primary logic gates one by one.

- **AND Gate:** There are two or more input signals in AND gate. The output of AND gate is TRUE only if all input variables are TRUE otherwise the output is FALSE. It performs logical multiplication i.e. AND function. As now we are considering only 2 input variables, so the truth table will have 2^2 = 4 combinations at the input side. To generalize, the input combinations in the truth table are dependent on the 'n' number of input signals as 2^n.

Y = A.B



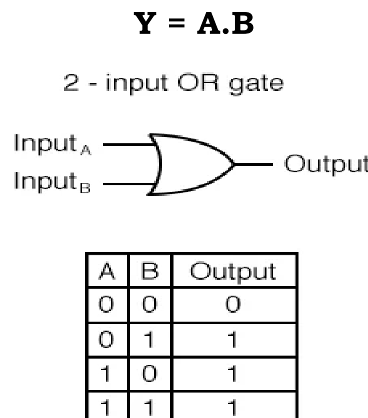
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2.2 Symbol and Truth Table of AND gate

(Source- https://www.allaboutcircuits.com/textbook/digital/chpt-3/multiple-input-gates/)

- **OR Gate:** In an OR gate, the number of input signals can be two or more and output is only one, just like in And gate. The output is TRUE if any of the input is TRUE otherwise it is FALSE. Logical addition is performed i.e. OR function.

Y = A.B

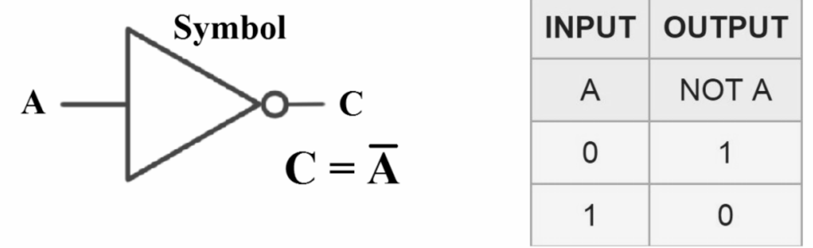


A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

Figure 2.3 Symbol and Truth Table of OR gate

(Source- https://www.allaboutcircuits.com/textbook/digital/chpt-3/multiple-input-gates/)

- **NOT gate:** It is one input and one output logical gate that gives the inverse outcome of the input. It performs inversion or complementation of the given input signal i.e. if the input is TRUE the outcome of NOT fate will be FALSE and vice versa.



INPUT	OUTPUT
A	NOT A
0	1
1	0

Figure 2.4 Symbol and Truth Table of NOT gate

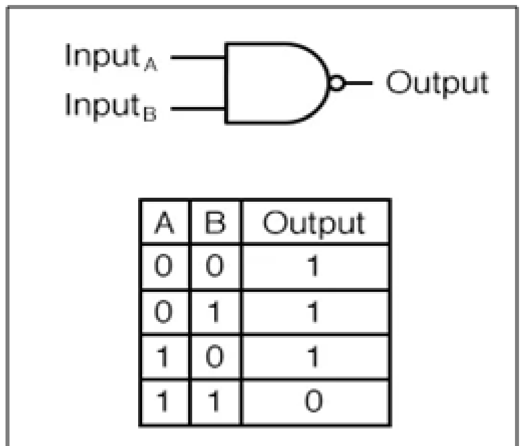
(Source- https://projectiot123.com/2019/05/24/introduction-to-not-gate/)

2.3.2 Universal Gates

Both NAND and NOR logical gates are termed as universal logic gates. It is due to the fact that all other logic gates can be accomplished by using either of

these two gates. Let's discuss them with their symbolic representation and truth table.

- **NAND gate:** It is designed by combining AND gate and NOT gate. As it is the inverse of AND gate, so the output of the NAND gate is FALSE when all inputs are TRUE otherwise it is TRUE.

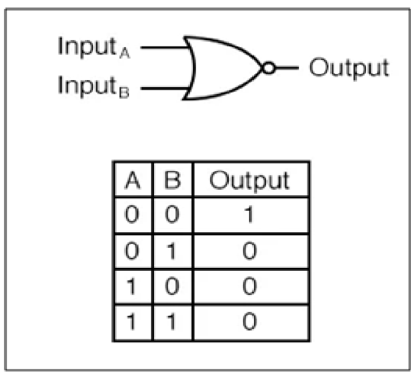


A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Figure 2.5 Symbol and Truth Table of NAND gate

(Source- https://www.allaboutcircuits.com/textbook/digital/chpt-3/ttl-nand-and-gates/)

- **NOR gate:** It is accomplished by the combination of OR and NOT gate. It provides the output as TRUE only when both the input signals are FALSE, otherwise, the output is FALSE. It generates a complement of the OR gate.

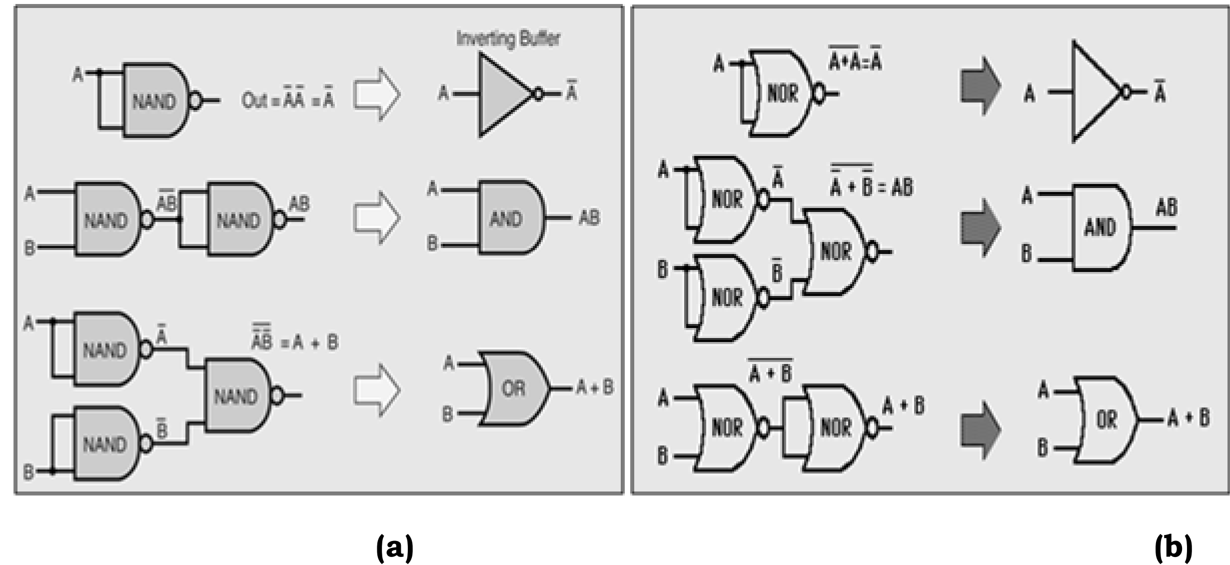


A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.6 Symbol and Truth Table of NAND gate

(Source- https://www.allaboutcircuits.com/textbook/digital/chpt-3/ttl-nor-and-or-gates/)

As it is already discovered that NAND and NOR gates are universal gates as they are the simplest logic gates to combine and generate all other basic gates. Figure 2.7 (a) & (b) shows how the basic gates are accomplished using these universal gates.



(a) **(b)**

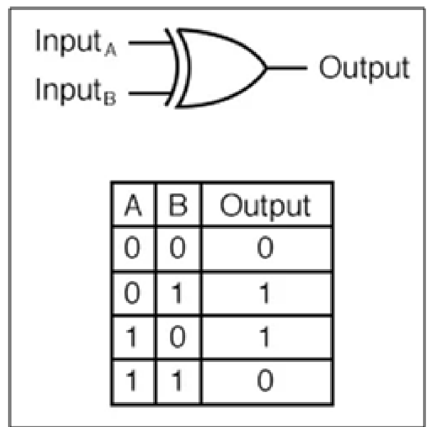
Figure 2.7 Generating other gates from NAND and NOR gates

(Source- http://hyperphysics.phy-astr.gsu.edu/hbase/Electronic.html)

2.3.3 Combinational Gates

The combinational gates are designed by combining the common (AND, OR, and NOT) and universal gates (NAND and NOR). There are two combinational gates Exclusive OR (XOR) and Exclusive NOR (XNOR).

- **XOR Gate**: It is also called the Exclusive OR gate. The output of the XOR gate is TRUE when either of the two inputs is TRUE, otherwise, it is FALSE. Generally, it is a 2 input and 1 output gate but when required it can be used for multiple inputs, such as multiple XOR gates that can be used in combination. The output of such multiple XOR gate will depend on the number of TRUE outcomes, the result will be ‘1’ when the number of 1s in the input is odd and it is ‘0’ when the number of 1s in the input is even.



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.8 Symbol and Truth Table of XOR gate

(Source- https://www.allaboutcircuits.com/textbook/digital/chpt-3/multiple-input-gates/)

The XOR gate can also be designed using the basic gates. Figure 2.9 illustrates the equivalent circuit of the XOR gate using AND, OR, and NAND gate in two ways:

1. $A \oplus B = AB + B\bar{A}$
2. $A \oplus B = (A+B)\ (AB)$

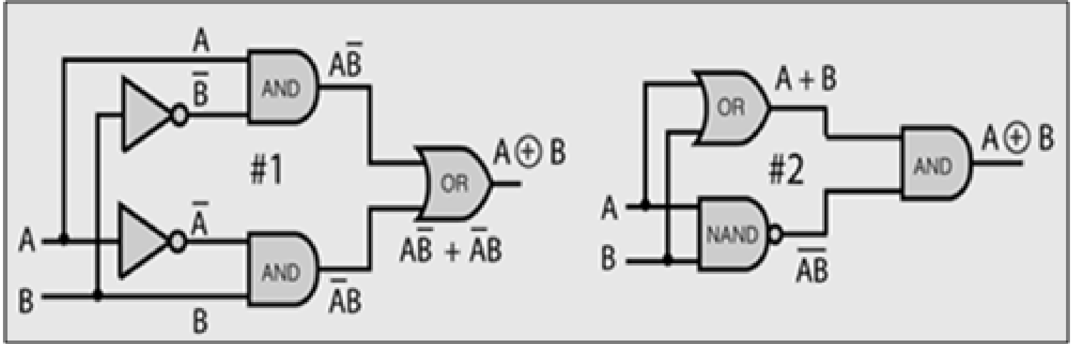
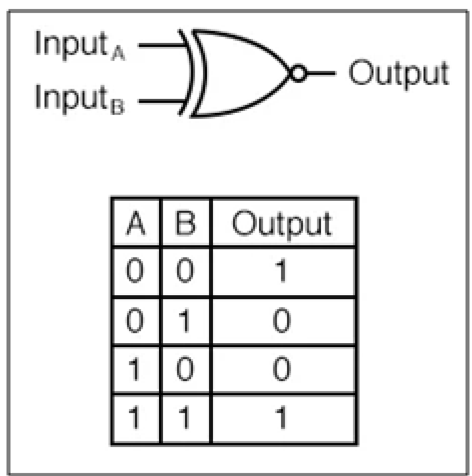


Figure 2.9 Equivalent circuit of XOR using other logical gates

(Source- http://hyperphysics.phy-astr.gsu.edu/hbase/Electronic/xor.html)

- **XNOR Gate:** The Exclusive NOR (XNOR) gate can be accomplished with the help of XOR followed by NOT gate. The output of XNOR is TRUE when the input signals are the same and FALSE when the inputs are different.



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

Figure 2.10 Symbol and Truth Table of XNOR gate

(Source- https://www.allaboutcircuits.com/textbook/digital/chpt-3/multiple-input-gates/)

Like the XOR gate, the XNOR gate can also be designed using basic gates, as shown in figure 2.11.

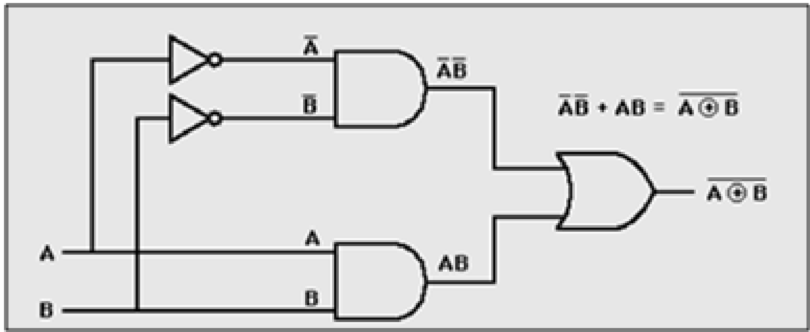


Figure 2.11 Equivalent circuit of XNOR using basic gates

(Source- http://hyperphysics.phy-astr.gsu.edu/hbase/Electronic/xnor.html)

2.4 K-Map Simplification

As the internal circuitry of the computer system is based on the Boolean algebra but it is much more complex to implement and execute. Complex logic diagrams and Boolean functions are the results of complex algebraic expressions of the circuits. The functionality of the algebraic functions may differ as they appear. The truth table is a unique feature for every function.

A simple approach to simplify these Boolean expressions is the Map Simplification method. It is the procedure of representing the truth table in a

pictorial form and selecting the minimum terms required to express a particular Boolean function. Another name for the map method is ***Karnaugh map or K-map***. Due to this minimization in the number of terms, there will be less number of logic gates and lesser number of variables in a Boolean expression. This will also ultimately result in reduced cost and power consumption of the computer system.

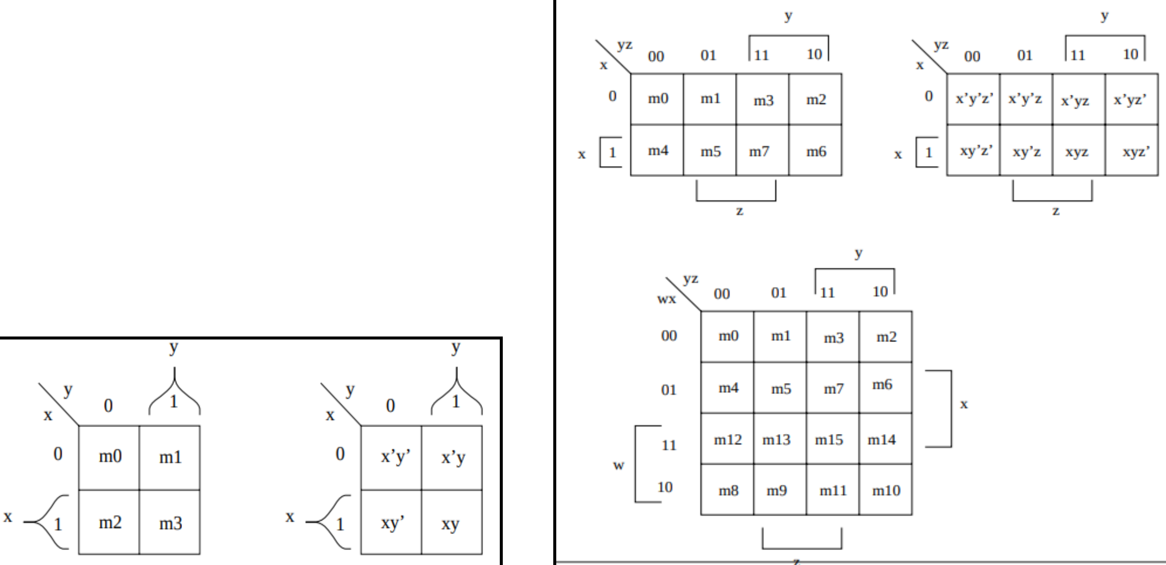


Figure 2.12 K-map for 2,3, and 4 variables

(Source- http://osp.mans.edu.eg/cs212/CS212_chapter_3_notes.pdf)

After simplification of a Boolean function using K-map, the obtained result is in a specified form. There are two fundamental forms of a logic function. A logical expression can occur as ***'Sum of Products (SOP)'*** or ***'Product of Sums (POS)'.*** SOP and POS are complementary to each other.

- **Sum of products (SOP) simplification:** In this form, the sum terms perform the OR operation and the product refers to the AND operation of these terms. It will be clearer with the help of an example. Let us consider the truth table for 3 input variables A, B & C generating the output Y=1 when the variables are ANDed for the condition of rows 2,3,5,7. In any of the four cases, the output will be 1, so as a cumulative result, these conditions can be ORed.

Row	A	B	C	Y
1	0	0	0	0
2	0	0	1	1
3	0	1	0	1
4	0	1	1	0
5	1	0	0	1
6	1	0	1	0
7	1	1	0	1
8	1	1	1	0

Then the required equation will be,

$$Y = \bar{A}\,\bar{B}\,C + \bar{A}\,B\,\bar{C} + A\,\bar{B}\,\bar{C} + A\,B\,\bar{C}$$

Each term in the above equation corresponds to minterms and is expressed using the symbol $\sum \boldsymbol{m}$.

- **Product of Sums (POS) Simplification:** In this form, the product terms perform the AND operation and the sum refers to the OR operation of these terms. Each term in POS form is referred to as maxterm and is denoted as ***ΠM.*** For example,

Some Major features of K-Map are:

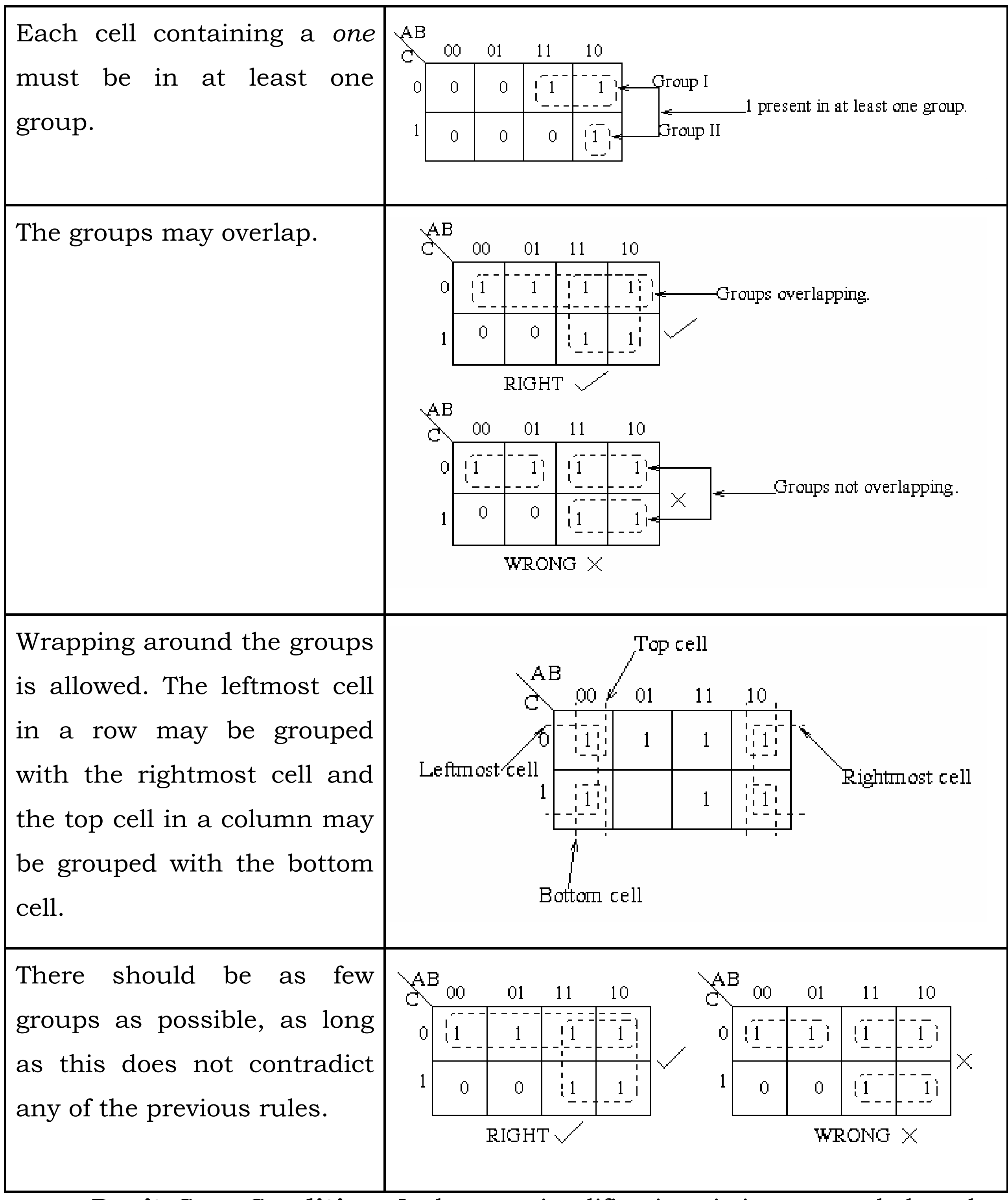
- In the SOP form, the combination of variables is called a minterm, which is represented in a truth table. The K-map is equipped with a possible minterm in each cell. For example, the truth table of the Boolean expression shown in figure 2.10 for the XOR gate contains four minterms. Generally, if a Boolean function is expressed by using n variables it will have 2^n minterms. The information in a truth table can be compacted by only listing the minterms that give the output '1'. For example, in figure 2.10, the Boolean expression for the XOR gate can be expressed as:

$$F\ (A, B) = \sum \mathbf{m}\ (0, 4)$$

  The minterms (0, 4) refer to the binary variables of the truth table of XOR gate. The symbol ∑ denotes the sum of the minterms. The minterms which are not included in the above equation represent that the output of such variables are '0'.
- The K-map consists of squares, each square represents one minterm. The minterms with output '1' are marked as '1' and others are marked as '0' or left empty. Various combinations and patterns of the squares marked as '1' are generated and they result in alternative algebraic expressions for the given Boolean function.
- ***Grouping:*** If there are two adjacent 1's in the squares of K-map then grouping can be done to simplify the equation. Similarly, four 1's can also be grouped. To generalize, 1's can be grouped only in multiples of 2. Another grouping can be done for the corners, if there are 1's in every corner then, a group can be formed by rolling up the lower 1's and forming the group of four 1's. The priority of grouping can be set accordingly to cover the maximum number of 1's in the map. Table 2.2 shows the grouping rules for K-map simplification.

Table 2.2 Rules for grouping in k-map simplification

Rules	**Example**
Groups may not include any cell containing a zero	WRONG × / RIGHT ✓
Groups cannot be diagonal, can be vertical or horizontal.	WRONG × / RIGHT ✓
A group can only be formed for 2^n cells, n= 1, 2, 3. If n = 1, a group will contain two 1's since $2^1 = 2$. If n = 2, a group will contain four 1's since $2^2 = 4$.	Group of 2 RIGHT ✓ / Group of 3 WRONG × / Group of 4 RIGHT ✓ / Group of 5 WRONG ×
A group should be as large as possible to cover up maximum cells.	RIGHT ✓ / WRONG × (Note that no Boolean laws broken, but not sufficiently minimal)



Rule	Example
Each cell containing a *one* must be in at least one group.	
The groups may overlap.	
Wrapping around the groups is allowed. The leftmost cell in a row may be grouped with the rightmost cell and the top cell in a column may be grouped with the bottom cell.	
There should be as few groups as possible, as long as this does not contradict any of the previous rules.	

- **Don't Care Condition:** In k-map simplification, it is assumed that the cells which do not have output as 1 can be marked as 0. But, there are certain cases in which there are no combinations of the input variables. In such a situation, these cells can be marked as 0 or 1. Such conditions

are referred to as *don't care conditions* and denoted as ***'x'.*** They can be considered as 0 or 1 according to the requirement to form the proper groups in the k-map.

2.5 Combinational Circuits

In the above section, we have studied logic gates which form the basis of various types of circuits in digital electronics. The combinational circuits are formed by the combination of basic and other gates. The output of combinational circuits depends upon the combination of inputs and does not change dynamically due to some clock input. These circuits have 'm' outputs for 'n' input signals as shown in figure 2.12. Some combinational circuits are represented in figure 2.7 (a) & (b) where universal gates NAND and NOR are used to generate the basic gates. Decoder and Multiplexer are other examples of the combinational circuits.

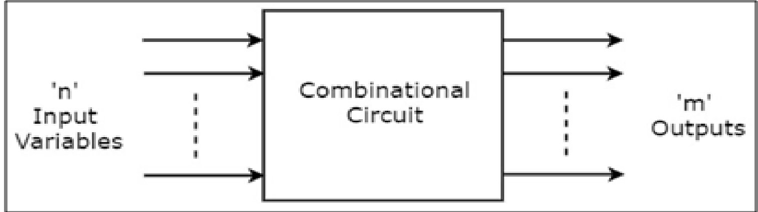


Figure 2.12 Combinational Circuit with *n* inputs and *m* outputs

(Source- ttps://www.tutorialspoint.com/digital_circuits/digital_combinational_circuits.htm)

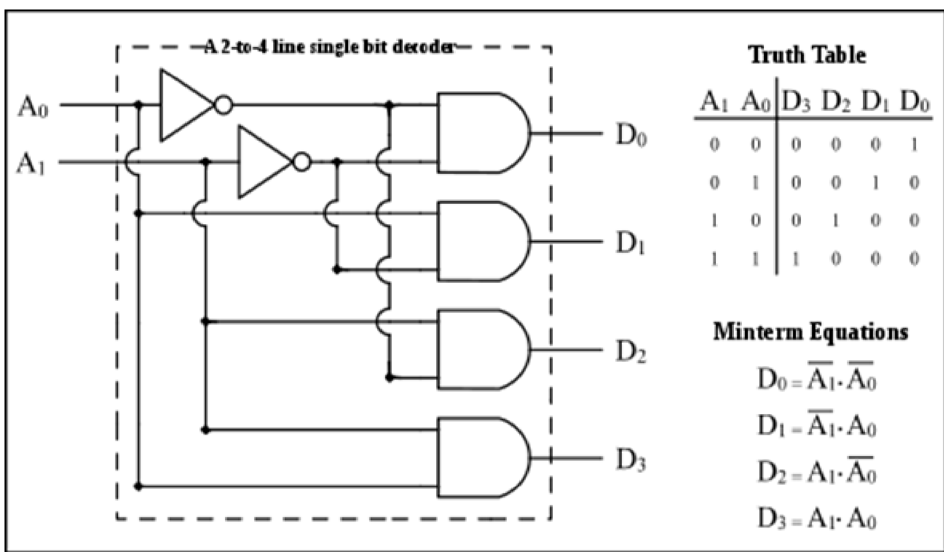
To design a combinational circuit, the following procedure is followed:

1. The given specifications decide the number of inputs and outputs.
2. Generate **Truth Table**. For 'n' inputs '2^n' combinations are possible. With the help of the truth table, output for corresponding input can be obtained.
3. **Boolean expression** is generated for each output and it is simplified.
4. Using **logic gates**, the simplified Boolean expression is implemented in a circuit form.

- **Decoder**

While transmitting any information, the data is encoded into certain codes and at the receiver's end; it is decoded to be understandable by the receiver. A decoder is a device that is used to decode the codes at the receiver end. The basic circuit of a decoder is obtained from combinational circuits, having 'n' input lines, and '2^n' output lines. Whenever the decoder is enabled, it will select one output for the combination of inputs i.e. it detects a particular signal.

The binary decoders of type 'n to 2^n' are the combinational circuits that modify the binary data from 'n' coded inputs to '2^n' outputs. For example, 2 to 4 line decoder, 3 to 8 line decoder, or 4 to 16 line decoder. Figure 2.13 shows the simplest 2 to 4 decoder with 2 input lines A_0 and A_1, and 4 outputs drawn with the combination of the inputs D_0, D_1, D_2, and D_3. The truth table for the decoder indicates that if either both or any input is high (1) then the decoder will generate one output at a time. The minterm equations indicate Boolean expression for each input combination and the equivalent circuit represents the use of AND gate and NOT gate to form a decoder.



A_1	A_0	D_3	D_2	D_1	D_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

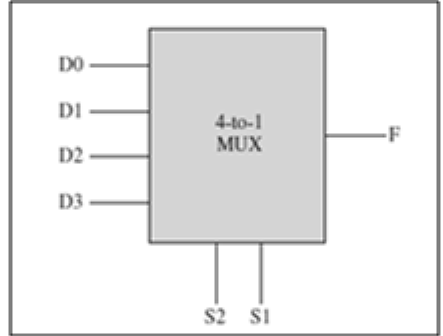
Figure 2.13 Truth table, Boolean expressions, and Equivalent Circuit for 2 to 4 decoder

(Source- https://en.wikipedia.org/wiki/File:Decoder_Example.svg)

- **Multiplexer**

A multiplexer is a switching combinational circuit being widely used in the field of digital electronics. It is a multiple-input and only one output device, generally used as a data selector. It is memory-less and no feedback path is observed in a multiplexer. A **MUX** (short term for Multiplexer) can be constructed using either traditional transistors like MOSFETs and relay switches or using high-speed logic gates to switch the voltage and binary data as input respectively through a single output. Generally, a multiplexer has ‘2^n’ input lines and ‘n’ *select/ control lines* and one undistorted output. The select lines help in selecting the particular input at a time. Combinations of input signals and select lines are formed and with the help of truth table and Boolean expressions, we are able to generate the equivalent circuit of the multiplexer using required logic gates. In the field of Computer architecture, the multiplexer is very important in data communications through the buses.

Figure 2.14 represents the basic 4-to-1 Multiplexer having 4 inputs D0, D1, D2, and D3, F is the output, and S1 and S2 are select lines. The circuit is obtained by using basic logic gates. NAND and NOR gates can also be used to determine the circuit of a multiplexer. Here, we can see that every select line whether ‘high’ or ‘low’ gives an input combination as the output. Besides this, we can also construct 8-to-1, 16-to-1 multiplexers, and so on.



S2	S1	F
0	0	D0
0	1	D1
1	0	D2
1	1	D3

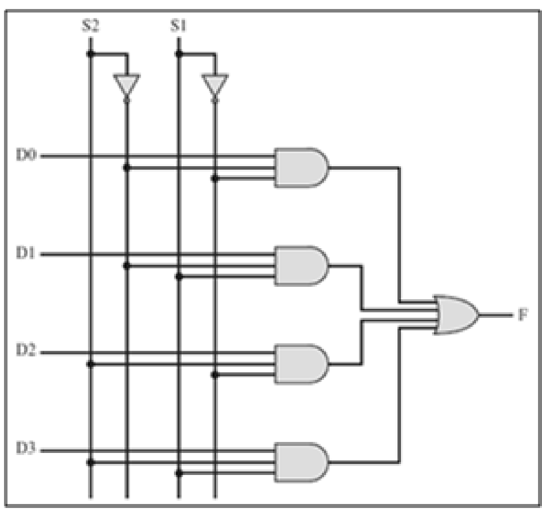


Figure 2.14 4-to1 Multiplexer block diagram, truth table, and circuit diagram

(**Source-** Computer Organization and Architecture, Ninth Edition, William Stallings, Chapter-11, Page no. 380)

2.6 Arithmetic Circuits

The circuits that are capable of performing arithmetic operations in any digital system are called Arithmetic Circuits. They are also sort of combinational circuits. The primary operation is addition, which forms the basis for other arithmetic operations like subtraction, division, and multiplication. All these operations will be discussed in upcoming units; here we are going to discuss the adder circuits in the digital system.

The binary addition is different from the Boolean algebra. We all are aware of the fact that the basic mathematical addition generates a 'carry' after performing the operation. Similarly, in binary addition also, a carry is generated but it is not so in case of Boolean algebra. The basic rules for binary addition are:

- 0+0 = 0
- 0+1 = 1
- 1+0 = 1
- 1+1 = 0 and carry 1 (the binary number 10)

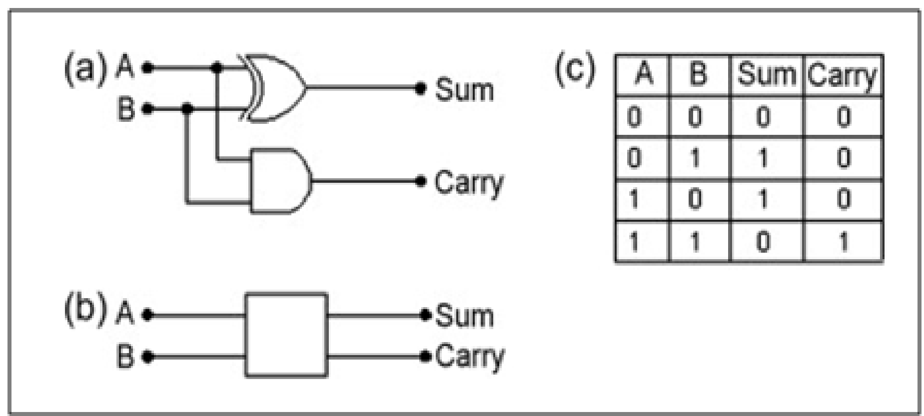
There are two basic addition circuits developed in the binary system, Half-adder and Full- adder. Let's have a look at the specifications of both circuits.

- **Half- Adder**

  It is the simplest adder circuit of a binary system that allows the addition of two bits, generating the output sum and a carry bit. The half-adder circuit is constructed by combining an **XOR** and an **AND** gate. The carry bit is '1' only when both the bits are '1' otherwise; it is '0', as shown in figure 2.15.

  The Boolean expression for half-adder can be:

  Sum = $A \oplus B$ and **Carry (C_{out}) = A.B**



A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure 2.15 Half- adder circuit, block diagram, and truth table

(**Source-** https://www.sciencedirect.com/topics/engineering/arithmetic-circuit)

- **Full- Adder**

  In a binary system, half-adder is sufficient to add two bits and it also produces a carry-out. But if the addition of more than two bits is to be considered, then the carry generated can't be ignored. The full- adder generates carry out and simultaneously uses this carry again as an input. That means, in a full-adder circuit, there are three input signals, two of them are operands to be added and the third input signal is the Carry (C_{in}).

  The full-adder circuit can be constructed by combining two half-adder circuits with the help of an OR gate. Figure 2.16 represents the circuit

diagram and truth table for full-adder. The Boolean expression for full-adder can be denoted as:

Sum = $A \oplus B \oplus C_{in}$

$C_{out} = A \cdot B + (A \oplus B) \cdot C_{in}$

Or

$C_{out} = A \cdot B + (A + B) \cdot C_{in}$

C_{in}	A	B	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 2.16 Full- adder circuit, block diagram, and truth table

(**Source-** https://link.springer.com/chapter/10.1007/978-3-030-13605-5_12)

It is also a fact that a full-adder can be converted to a half-adder circuit if the carry input C_{in} is connected to a 0 voltage level permanently. To design an n-bit adder, full- adders can be cascaded together in such a way that carry is forwarded from full-adder A to full-adder B and sum is generated on each full-adder circuit but it is dependent on the carry bit from the previous circuit, as shown in figure 2.17. It is also known as Ripple Carry Adder.

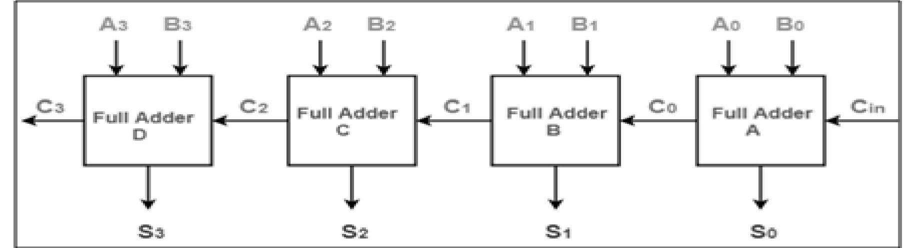


Figure 2.16 Block diagram of 4-bit adder using full-adders in cascaded form

2.7 Sequential Circuits

The sequential circuits are dynamic in nature i.e. the output of such digital circuits is time-dependent and changes with time. They are used to provide a memory to the binary operations in the digital system. They are more complex circuits than the combinational circuits. The output of the sequential circuits is dependent on the current as well as previous inputs. These circuits follow the concept of feedback and are said to be operated by a *clock signal* that is generated using a *clock generator circuit*. The clock pulse operates between 0 and 1 level. The circuit operates when the clock is at level 1. The clock signal plays an important role in sequential circuits to make them operate in *Synchronous* mode.

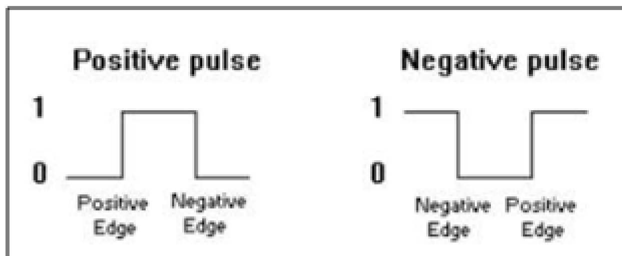


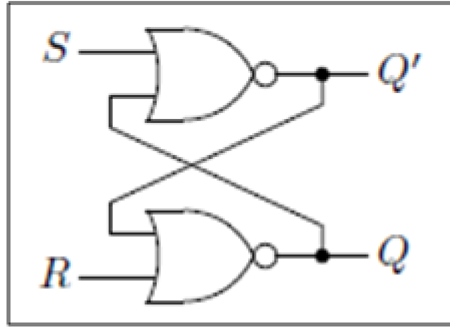
Figure 2.17 Clock Pulse

(**Source-** http://www.circuitstoday.com/triggering-of-flip-flops)

2.7.1 Basic Latch

A latch is a storage device with the capacity to store one bit at a time. It can be constructed using two or more logic gates such that the output of one gate can be fed as input to the other and the output of this second gate can be given as feedback input to the first gate. Figure 2.18 represents the NOR gate circuit

implementation for the most fundamental latch i.e. **SR- Latch** (Set- Reset). The circuit has two inputs S (Set) and R (Reset), two outputs Q and Q′. The fact can be noted that the latch can remain in the same state until the next input is given to it, i.e. it can store the data till the next input. If Q = 1 (Q′ = 0), then the latch is *Set*, while if Q = 0 (Q′ = 1), then it is *Reset*. The circuit is said to be *bi-stable* as it has two stable states.



Inputs		Outputs	
S	**R**	**Q**	**Q′**
0	0	Q_0	Q'_0
1	0	0	1
0	1	1	0
1	1	X	X

Figure 2.18 Implementation of SR- Latch by using NOR gate and its truth table

(**Source-** Introduction to Computer Organization, Robert G. Plantz, Chapter- 5, Page no. 96)

The operation of SR- Latch is followed as four possible input combinations:

a) S = 0, R = 0 (Keep Current State)

- If Q = 0 and Q′ = 1, then the output of the upper NOR gate is 1 and that of lower NOR gate is 0.

- If Q = 1 and Q′ = 0, the output of the upper NOR gate is 0 and that of lower NOR gate is 1.

Thus, the state is maintained as SET or RESET, due to the cross feedback between both the gates.

b) S = 1, R = 0 (Set)

- If Q = 1 and Q′ = 0, the upper NOR gate gives the output 0 and the lower NOR gate gives 1. This means that the latch is in *Set* state.
- If Q = 0 and Q′ = 1, the upper NOR gate will give the output 0, which is given back to the lower NOR gate to produce the output as 1. This cross-feedback system maintains the output of the upper NOR gate at 0.

c) S = 0, R = 1 (Reset)

- If Q = 1 and Q′= 0, the lower NOR gate will produce the output of 0, therefore causing the upper NOR gate to produce the output as 1. The latch moves into the *Reset* state.
- If Q = 0 and Q′ = 1, the output of the lower NOR gate is 0 and that of the upper NOR gate is 1. Thus, the latch remains in the Reset state only.

d) S = 1, R = 1 (Undefined state)

- If Q = 0 and Q′ = 1, the output of the upper NOR gate (0) is given back as an input to the lower NOR gate, giving the output as 0. Thus, the condition appears when Q = Q′ = 0, which is an undefined state.
- If Q = 1 and Q′ = 0, the output of the lower NOR gate (0) is given as an input to the upper NOR gate to give output 0. Again the same condition appears Q = Q′ = 0, which is an undefined state.

2.7.2 Flip- Flop

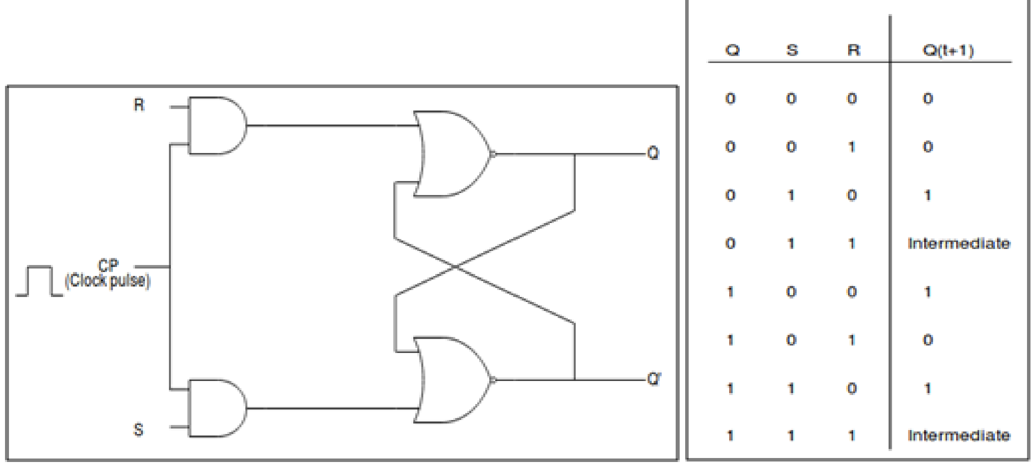
A flip- flop is the combination of latches and changes its output with the clock pulse to provide synchronous output. It has 1-bit memory like a latch. The difference between a latch and a flip-flop is in terms of a clock signal. Flip- flop is regulated according to the clock pulse and it produces synchronous output, while latch doesn't depend on the clock pulse. We will now discuss the different

types of flip-flops that are most widely used in the digital circuits and in computer hardware.

Clocked S-R Flip Flop

The introduction of the clock signal in the computer circuits has become essential as all the operations of the computer are synchronized with the clock. The clocked SR flip- flop overcomes the undefined state problem of the SR latch. Figure 2.19 shows the circuit for clocked SR Flip- flop which is an extended form of SR latch with two inputs and two outputs, controlled by a clock pulse. The S and R inputs are valid till the clock pulse is high and as soon as the clock pulse becomes low, old outputs Q and Q will remain unchanged and the flip-flop will wait for the next input when the clock pulse will become high.

The major drawback of SR flip-flop is the intermediate state, which is undesirable and occurs when both the inputs S and R are high (1). The output can't be predicted in such a condition.



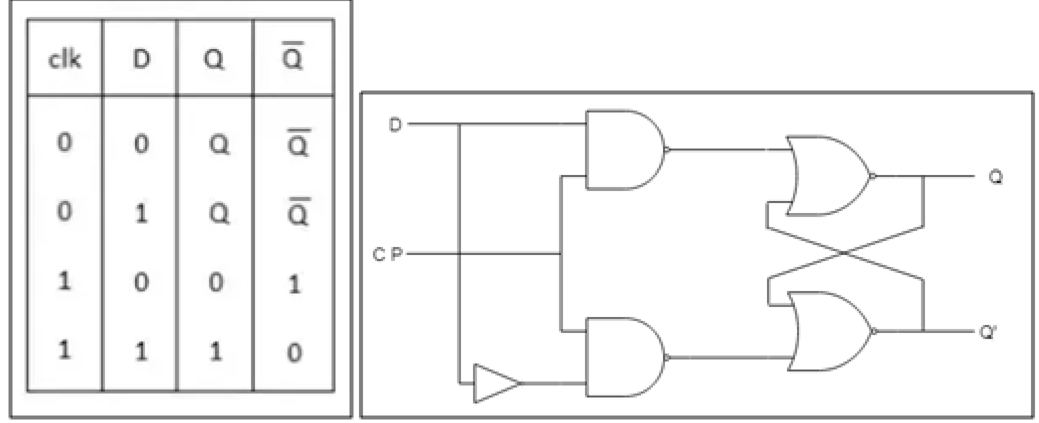
Q	S	R	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	Intermediate
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	Intermediate

Figure 2.19 SR Flip- flop Circuit representation and truth table

(**Source**- https://www.javatpoint.com/s-r-flip-flop)

D Flip- flop

The D flip- flop overcomes the problem of clocked SR flip-flop in which both the inputs S & R cannot be high simultaneously. In D flip- flop, R input is the inverted form of input S that is obtained by putting a NOT gate between S & R inputs as represented in figure 2.20.



clk	D	Q	$\overline{Q}$
0	0	Q	$\overline{Q}$
0	1	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

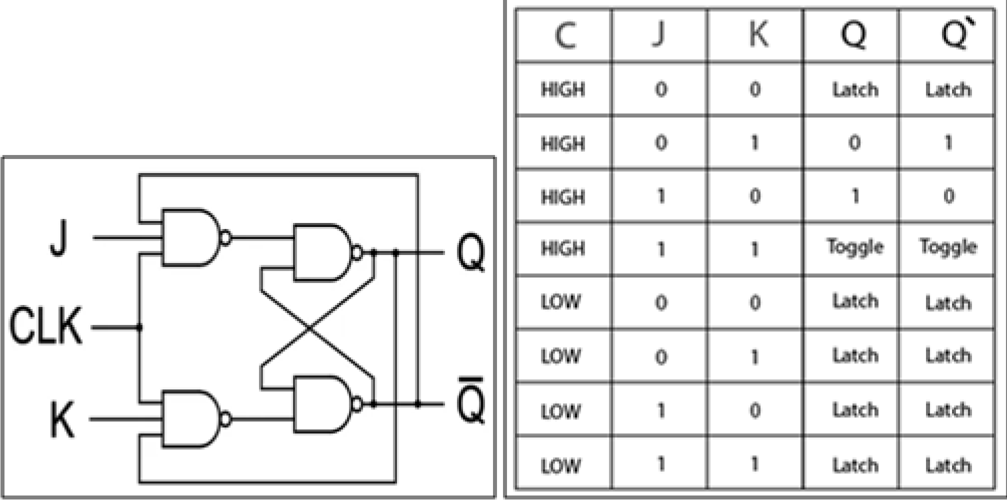
Figure 2.20 D Flip- flop truth table and a circuit representation

(**Source**- https://www.javatpoint.com/d-flip-flop)

When the clock pulse is high, the flip- flop is in its SET state and if it is low, the flip- flop adopts a CLEAR mode.

JK Flip –flop

JK flip- flop is a modified version of SR flip- flop. The intermediate state of SR flip- flop is eliminated in JK flip-flop. As discussed, when both the inputs in SR flip- flop is high, an intermediate state appears which is eradicated in JK flip-flop. It is done by using an additional feedback system to the SR flip- flop as shown in figure 2.21. In JK flip-flop, when both the inputs J & K are high (1), then the output keeps on toggling itself between the two states using the NAND gates. Toggling of outputs means when Q =1, it switches to Q = 0 and when Q=0, it switches to Q = 1.



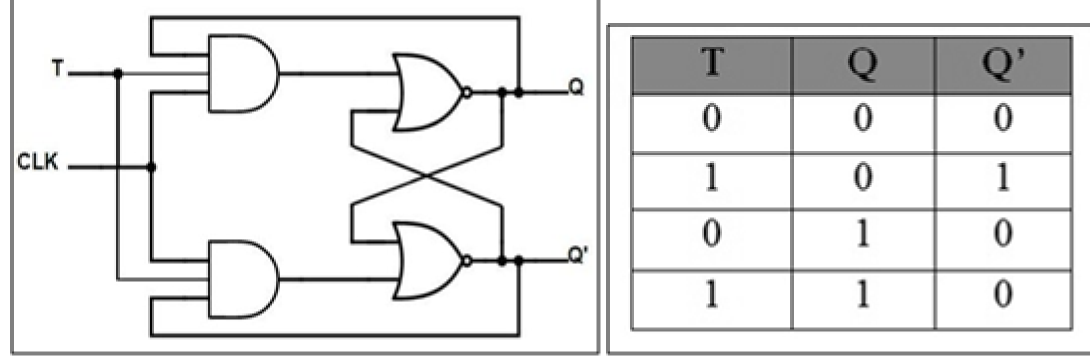
C	J	K	Q	Q`
HIGH	0	0	Latch	Latch
HIGH	0	1	0	1
HIGH	1	0	1	0
HIGH	1	1	Toggle	Toggle
LOW	0	0	Latch	Latch
LOW	0	1	Latch	Latch
LOW	1	0	Latch	Latch
LOW	1	1	Latch	Latch

Figure 2.21 JK Flip- flop circuit representation and truth table

(**Source**- https://dcaclab.com/blog/j-k-flip-flop-explained-in-detail/)

<u>T Flip- Flop</u>

T flip- flop is a single input JK flip- flop in which both J & K inputs are connected to each other. It is also known as toggle flip- flop as they have the ability to complement or toggle the states. Figure 2.22 represents the circuit and truth table of T flip- flops.



T	Q	Q'
0	0	0
1	0	1
0	1	0
1	1	0

Figure 2.22 T Flip- flop circuit representation and truth table

(**Source**- https://dcaclab.com/blog/j-k-flip-flop-explained-in-detail/)

2.8 Registers and Counters

As discussed, flip- flops are the building blocks of various digital circuits in the computer system. They are used in different arrangements and formats to form the circuits for different functions. Registers and counters are the basic

applications of flip- flops being used in the computer circuits. Let’s discuss registers and counters in detail.

- **Registers**

  A register is a storage device consisting of a group of flip- flops in the CPU and is capable of storing one or more bits of binary data. To generalize, an *n-bit* register is formed by combining *n flip- flops* and can store *n-bits* of binary data. Along with the flip-flops, combinational logic gates are also combined to form the register circuit. The purpose of the flip- flop is to store the information while the gates control the transferring of new information in the register. Figure 2.23 represents a 4-bit register constructed using four D flip-flops. All the four flip-flops are triggered at the rising edge using the same clock pulse. The CLEAR input is fed in a special terminal of each flip-flop. When the input is 1, all flip- flops get reset asynchronously. Prior to the clock operation, the clear input clears the registers to all zeroes and it is independent of the clock signal. The process of feeding new information into the register is called *loading* the register.

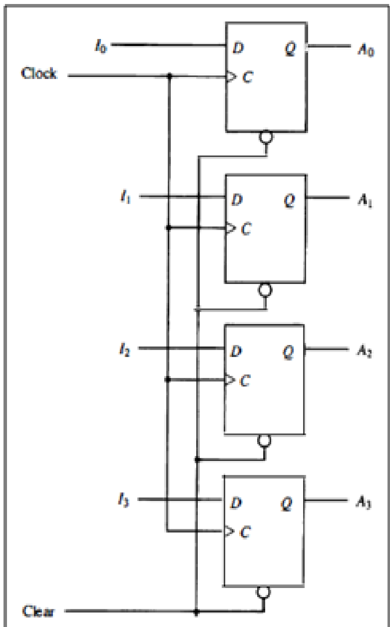


Figure 2.23 4- bit register using D flip-flop

(**Source**- Computer System Architecture, Morris Mano, Chapter- 2, Page- 51)

Registers are of two types: Parallel register and a Shift register.

Parallel Register: The registers in which data can be read and write simultaneously are called parallel registers. A variety of parallel registers are used in the computer system to store the data temporarily. They can be of 8 or 16 bits, depending upon the need of the function of the CPU. The register arrangement shown in figure 2.23 also depicts a parallel register, in which different inputs are fed in a parallel format to all the four D flip- flops and are governed by a single clock pulse.

Shift Register: These types of registers are generally used in the serial transmission of binary data. The data is fed to the first flip- flop and it is shifted to the next flip- flop with each clock pulse. Shift registers can also be used as an interface in serial input/ output devices. ALU also requires shift registers to shift data after performing the operation. Some shift registers are capable of shifting the data in one or both directions. As per the requirement of the operation, the shift registers can be classified in 4 types: Serial in Serial out (SISO) shift register, Serial in parallel out (SIPO) shift register, Parallel in serial out (PISO) shift register, and Parallel in parallel out (PIPO) shift register. Figure 2.24 displays a 4-bit shift register comprising of four D flip-flops, operating as serial in serial out (SISO).

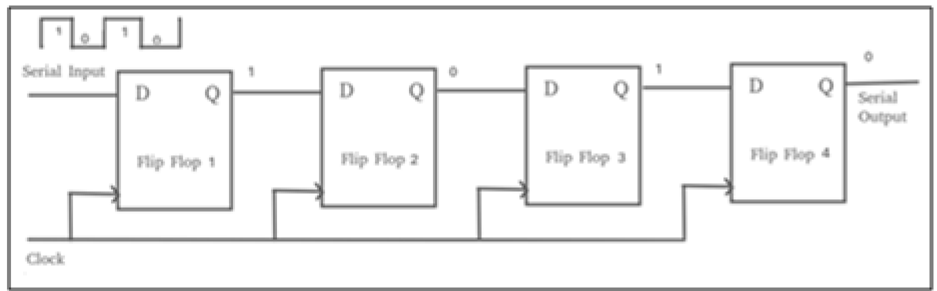


Figure 2.24 4- bit shift register (SISO)

(**Source**- https://www.geeksforgeeks.org/shift-registers-in-digital-logic/)

- **Counters**

Counters are types of sequential circuits that work on the concept of sequence of states as per the inputs. They are common in almost all

digital circuits. They are used to count the occurrence of any operation in the computer system. The counter that involves a binary data sequence is called a *binary counter*. The n- bit binary counter consists of registers containing n flip-flops and logic gates to count n- bits of the operation (0 to 2^n-1). The counter increments its value each time it counts one bit which indicates that the system is ready for the next bit. The basic example of a counter is a Program counter in the CPU of a computer.

The basic circuit of a counter comprises generally T or JK flip- flops due to their complementing property. On the basis of their way of operations, counters can be synchronous or asynchronous. The synchronous counters are fast in comparison to the asynchronous counters, as all the flip- flops in the synchronous mode change their state on application of one clock pulse; this increases their use in the computer systems. A *Ripple Counter* is a type of asynchronous counter, implemented using JK flip- flops.

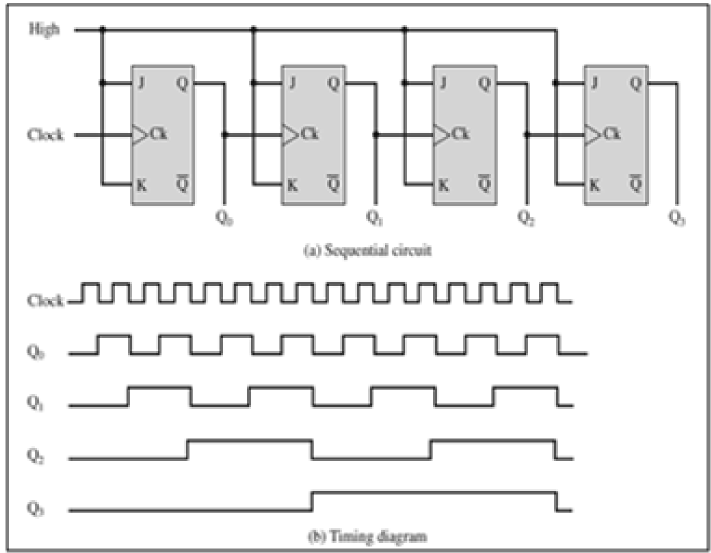


Figure 2.25 4- bit Asynchronous Ripple Counter with the timing diagram

(**Source**- Computer organization and architecture, William Stallings, Chapter- 11, Page no.- 394)

Figure 2.25 represents a 4-bit ripple counter using JK flip-flops along with the timing diagram showing a delay in the output of the last flip- flop. In the ripple

counter, only the first flip-flop is being clocked by the external clock pulse, and other flip- flops are clocked with the output of the first flip-flop. Due to this rippling of the clock pulse, it is termed as a ripple counter.

On the other hand, synchronous counters overcome the drawback of the ripple counter, delay in the output of the counter. In synchronous counters, all the flip-flops are clocked by the single clock pulse at the same time as displayed in figure 2.25, a 4-bit synchronous counter. The inputs J & K maintain at 0 when the count enable is 0 without changing the output of the counter. When the counter is enabled, the output of the first flip- flop is complemented. The other three flip-flops are complemented when the preceding flip- flop's output is 1 and count is enabled. The AND gates help in generating the logic for J & K inputs.

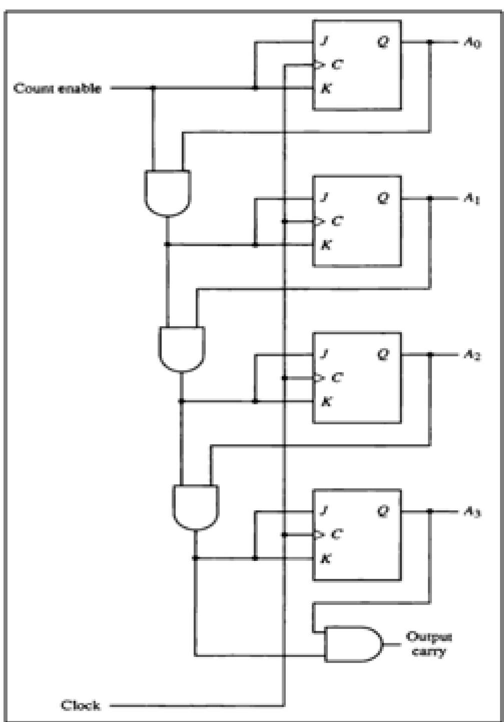


Figure 2.25 4- bit Synchronous Counter

(**Source**- Computer System Architecture, Morris Mano, Chapter- 2, Page- 57)

2.9 Summary

- Boolean algebra forms the building blocks for digital circuits, following the concept of two states 1 (high) and 0 (Low).

- Logic gates are basic fundamentals of the circuits. AND, OR and NOT are the primary logic gates while NOR and NAND are known to be Universal logic gates. XOR and XNOR are the other combinational gates.
- The combinational, arithmetic and sequential circuits are constructed using various combinations of logic gates and all have their distinct applications.
- Latches and flip-flops are used to store the data temporarily during the operation.

2.10 Key Terms

- **Boolean algebra:** It is the mathematical foundation that is used to design digital circuits and other digital systems and analyze their behavior and fundamental operations.
- **Decoder:** A decoder is a combinational circuit having n input lines and 2^n output lines.
- **Multiplexer:** A combinational circuit with 2^n input lines, n select lines, and only one output.
- **Synchronous:** The mode which is dependent on the clock pulse. It changes with the change in the clock signal.

2.11 Check Your Progress

Q1) State the different Boolean Operators and Identities.

Q2) Discuss the different types of logic gates with the help of circuit diagrams.

Q3) Explain the logic of half-adder and full- adder using a circuit diagram.

Q4) Differentiate between Synchronous and Asynchronous Counters.

Q5) Define: a) Latch b) Flip- flop c) Registers

Q6) Design XOR and XNOR gates using basic logic gates.

Q7) What is the basic difference between a latch and a flip-flop?

References:

Computer Architecture and Organization, Subrata Ghoshal, Pearson Publication.

Computer Organization and Architecture, 9th edition, William Stallings, Pearson Publication.

Computer System Architecture, M. Morris Mano.

http://epgp.inflibnet.ac.in/epgpdata/uploads/epgp_content/S000574EE/P001494/M015065/ET/1459848930et05.pdf

https://www.allaboutcircuits.com/textbook/digital/chpt-3/multiple-input-gates/

https://www.sciencedirect.com/topics/engineering/arithmetic-circuit

https://link.springer.com/chapter/10.1007/978-3-030-13605-5_12

https://www.edgefx.in/digital-electronics-latches-and-flip-flops/

MODULE: II

COMPUTER ARITHMETIC AND MICROOPERATIONS

Unit: 3 – Computer Arithmetic

Structure

3.0 Introduction

3.1 Unit Objectives

3.2 Data Representation

    3.2.1 Conversion Techniques

3.3 Addition and Subtraction of Binary Numbers

    3.3.1 Two's Complement Method

3.4 Multiplication of Binary Numbers

    3.4.1 Booth's Algorithm

3.5 Division of Binary Numbers

3.6 Floating-Point Number Representation

3.7 Floating-Point Arithmetic and Unit Operations

    3.7.1 Floating-point Addition and Subtraction

    3.7.2 Floating-point Multiplication

    3.7.3 Floating-point Division

3.8 Binary Codes and Error Detection Codes

3.9 Summary

3.10 Key Terms

3.11 Check Your Progress

3.0 Introduction

Computers are meant to perform distinct arithmetic operations like addition, subtraction, multiplication, and division. These computer arithmetic operations are subjected to two different types of numbers: integers and floating-point numbers. Some operations are hardware-based while others are accomplished using certain software programs and algorithms. The operations at the processor level follow binary representation while decimal representation of data is followed by the high- level language programs. The computer system is

dependent on four types of number systems, binary, decimal, octal, and hexadecimal. The base of the number represents its type. For instance, binary numbers have the base 2, base of decimal numbers is 10, octal numbers have the base of 8 and hexadecimal numbers have the base 16. There are certain techniques to convert one number to another number system by changing their base. Binary to decimal, decimal to binary, binary to octal, octal to binary, binary to hexadecimal, and decimal to binary, all types of conversion is possible as per the requirement. The computer system is based on the binary number system, so the input is given to the computer in any of the number systems. With the help of conversion techniques, the computer converts the input in binary form and then performs the desired operation to give the output. In this unit, we will discuss the various arithmetic operations and their techniques.

3.1 Unit Objectives

This unit will help the reader to gain knowledge about:

- Number System and various conversion techniques.
- Representation of signed and unsigned integers.
- Techniques used for the addition and subtraction of signed integers.
- Algorithm for multiplication and division of signed integers.
- Floating-point arithmetic operations.

3.2 Number System

A number system is the fundamental element of any mathematical calculations. It gives an idea about the type of calculation and what will be the output of such an operation. It is interesting to note that the very first computer ENIAC was based on the decimal number system, but later it was observed that the performance of the computer is better with the binary number system.

Decimal Number System

A decimal number contains ten numbers or symbols from 0 to 9. The relative position decides the weightage of the number. For example, the numbers 6 & 7 will have different weightage in numbers 76 and 67. The base of the decimal number is 10, which means every digit of the number is multiplied by 10^n (n=0 to 9), where n decides the position of the digit in the number.

For example, number 1587 will be represented as,

$$1587_D \text{ or } 1587_{10} = 1 \times 10^3 + 5 \times 10^2 + 8 \times 10^1 + 7 \times 10^0$$

To generalize the equation for all number systems, a number PQRS will be represented as,

$$\mathbf{PORS_{(base)} = P \times (base)^3 + Q \times (base)^2 + R \times (base)^1 + S \times (base)^0} \quad \ldots \quad \mathbf{(1)}$$

Binary Number System

The binary number system is based on two symbols 0 and 1 with base 2. It is considered to be the most suitable number system for computers as the circuits are designed using digital electronics. There are only two states in digital circuits i.e. on and off. So, this makes the binary system more accurate for computer operations. Equation (1) can be modified according to the binary number system as:

$$PORS_{(2)} = P^* (2)^3 + Q^* (2)^2 + R^* (2)^1 + S^* (2)^0$$

For example, the binary number $(1010)_2$ can be represented in decimal format as:

$$1010_2 = 1^* (2)^3 + 0^* (2)^2 + 1^* (2)^1 + 0^* (2)^0$$
$$= 8 + 0 + 2 + 0$$
$$= 10$$

Similarly, other decimal numbers can also be represented in binary form and vice versa. We will learn about the conversion techniques from binary to decimal and from decimal to binary afterward.

Octal Number System

Octal numbers have eight symbols (0 to 7) and the base of such numbers is also eight. Equation (1) for the octal number system can be written as:

$$PORS_{(8)} = P^* (8)^3 + Q^* (8)^2 + R^* (8)^1 + S^* (8)^0$$

Hexadecimal Number System

We know that all the computer operations are based on the binary number system but it becomes difficult for the user to understand the complex binary formats. The hexadecimal number system is used to form the interface between the computer operations and the user. This number system contains an alphanumeric series of 16 digits, 0 to 9 (numbers), and A to F (alphabets denoting numbers 10 to 15). It is understood that the base of the hexadecimal number is 16 and then modifying equation (1) according to the hexadecimal number system, we will obtain:

$$PORS_{(16)} = P^* (16)^3 + Q^* (16)^2 + R^* (16)^1 + S^* (16)^0$$

3.2.1 Conversion Techniques

There is a need for conversion of one number system to another. Different conversion techniques are defined for different number- systems.

- **Decimal to Binary Conversion**

  The decimal to binary conversion can be achieved by the method of successive division by 2. The remainder of each step of division is arranged in a format to obtain the binary equivalent. Let us study it with the help of an example. Let the decimal number to be converted into binary be 23.

2	23	1	↑ (LSB)
2	11	1	
2	5	1	
2	2	0	
	1		(MSB)

  So, the binary equivalent will be read from bottom (Most Significant Bit- MSB) to top (Least Significant Bit- LSB) for $\mathbf{(23)_{10} = (10111)_2}$. Similarly, the binary equivalent for other decimal numbers can also be calculated.

  For the numbers with decimal points, the conversion is simple. Both the parts are converted into binary separately. For instance, the binary number for (3.16) will be (11.00101000111). Here, 3 and 16 are

converted to binary with the same conversion method as discussed above, and then finally, both parts are combined to get the desired result.

- **Binary to Decimal Conversion**

To convert any binary number to decimal equivalent, we can directly use equation (1). For example, if we convert the above calculated binary number again into decimal equivalent, then by applying equation (1),

$$10111_{(2)} = 1 \times (2)^{4} + 0 \times (2)^{3} + 1 \times (2)^{2} + 1 \times (2)^{1} + 1 \times (2)^{0}$$

$$= 16 + 0 + 4 + 2 + 1$$

$$= (23)_{10}$$

Thus, the decimal equivalent for $\mathbf{(10111)_2 = (23)_{10}}$.

For the decimal point numbers, the power of the base 2 at the right part of the number starts with -1 and all the process of conversion remains the same. For example, to convert 101.011 to binary, then the equation will become:

$$101.011_{(2)} = [1 \times (2)^{2} + 0 \times (2)^{1} + 1 \times (2)^{0}] \,.\, [0 \times (2)^{-1} + 1 \times (2)^{-2} + 1 \times (2)^{-3}]$$

$$= (4 + 0 + 1) \,.\, (0 + 0.25 + 0.125)$$

$$= (5.375)_{10}$$

Thus, the decimal equivalent for $\mathbf{(101.011)_2 = (5.375)_{10}}$.

Decimal	Binary	Octal	Hex
0	00000000	000	00
1	00000001	001	01
2	00000010	002	02
3	00000011	003	03
4	00000100	004	04
5	00000101	005	05
6	00000110	006	06
7	00000111	007	07
8	00001000	010	08
9	00001001	011	09
10	00001010	012	0A
11	00001011	013	0B
12	00001100	014	0C
13	00001101	015	0D
14	00001110	016	0E
15	00001111	017	0F

Figure 3.1 8-bit Conversion table for decimal, binary, octal, and hexadecimal numbers

(**Source**- http://web.alfredstate.edu/faculty/weimandn/miscellaneous/ascii/ascii_index.html)

- **Binary to Hexadecimal Conversion**

  For binary to hexadecimal conversion, the binary digits are taken in a group of 4 bits, starting from the right side of the binary number and these groups are then converted into its hexadecimal equivalent displayed in figure 3.1. For example, we have to convert 10101 into hexadecimal. Then a group of 4 digits from the right end will be formed 0101 and another group is 0001 (we can place as many 0 bits to the left end). Now, referring to the conversion table in figure 3.1, 0101 is represented as 05 in hexadecimal and 0001 as 01. So, the hexadecimal equivalent for $(10101)_2$ is $(15)_{16}$.

- **Hexadecimal to Binary Conversion**

  For hexadecimal to binary conversion, each hexadecimal digit is converted to binary equivalent using the conversion table in figure 3.1. For example, $(15)_{16}$ in binary format will be written as 1- 0001 and 5- 0101, and on combining both digits the binary equivalent for $(15)_{16}$ will be 00010101 or 10101.

- **Binary to Octal Conversion**

  The conversion of a binary number to octal equivalent can be in two ways. First, the binary number can be converted to the decimal number system first and then to the octal system. Secondly, the grouping method can be used just like in binary to hexadecimal conversion. In binary to octal conversion, a group of 3 digits is formed from the right end and then they are compared to the equivalent octal code given in the conversion table. For example, to convert 10010110 in octal form, first, it is divided into groups of 3 bits from the right end. 010 010 110, then from figure 3.1, it will be represented in octal form as $(2\ 2\ 6)_8$.

- **Octal to Binary Conversion**

The octal to binary conversion is similar to the conversion of hexadecimal to binary. Each digit is separately converted to binary equivalent using the table in figure 3.1. For example, to convert $(65)_8$ into the binary equivalent, a separately binary equivalent of 6 is 110, and 5 is 101. So, on combining both digits, $(65)_8$ in binary will be 110101.

3.3 Addition and Subtraction

The fixed-point numbers represent integers and fractions. The negative numbers can be signed or unsigned and slightly different arithmetic is followed for the signed numbers. We are already aware of the half- adder and full- adder circuits used to perform the addition of two or more bits. Half- adder circuits are not capable of taking account of the carry generated in addition operation while in full- adder circuits the carry is taken again as an input C_{in}. The binary addition is similar to the normal mathematical addition, but the operation of subtraction is different as integers represent signed and unsigned both values. Two's complement method is considered to be the most prominent method for addition and subtraction in computer arithmetic.

Before proceeding to the methods for the addition and subtraction of binary numbers, let us be familiar with the ***Signed magnitude representation***. In general mathematics, the positive and negative integers were represented by putting + or – signs in front of them. Here, for binary numbers, the positive and negative numbers are represented using the ***sign bit***. The leftmost bit i.e. most significant bit (**MSB**) is treated as the Sign bit. If the sign bit is 0, then the number is regarded as positive and if the sign bit is 1, then the number is said to be negative. For example, the decimal number 12 sign- representation is shown below:

+12 = 00001100

-12 = 10001100 (Due to Sign Representation)

One of the major drawbacks of signed magnitude representation is that both the signs of a number are taken into account for performing any operation on that number. Also, zero has two representations in signed magnitude form.

This results in inconvenience in performing operations using zero as a number. Due to this fact, 2's complement method is considered to be more convenient.

+0 = 00000000

-0 = 10000000 (Due to Sign magnitude)

3.3.1 Two's complement Method

The complement of a binary number is of two types: 1's complement and 2's complement. The 1's complement of any binary number is obtained by interchanging 0's and 1's in that number. For example, the 1's complement of binary number 101101 will be 010010. The 2's complement of any binary number is achieved by adding 1 to least significant digit (LSB) at the right end of the 1's complement. For example, the 2's complement of number 101101 will be given as:

```
101101 ———→ 010010 (1's complement)
               +1
            ------
            010011   (2's complement)
```

The 2's complement method was required to simplify the subtraction operation in digital computers. For a signed binary number, the 2's complement for positive numbers is the same as for unsigned numbers. The difference is in the negative numbers. First, the negative number is represented with a positive sign and then 2's complement of the number is taken. For example, to represent +7 and -7 in binary form, +7 will be simply denoted in signed representation as 00111(Sign bit is 0) and for -7 the following steps will be considered:

- +7 = 0 0111
- 2's complement of 7 will be 1 1001. Here, the sign bit is 1 which shows that the number is negative.

<u>Subtraction using 2's complement method</u>

In the subtraction operation, the first number is minuend and the other number is subtrahend (the number that is to be subtracted from minuend). For the subtraction of two binary numbers, the following algorithm can be used:

- Obtain 2's complement of the subtrahend.
- Add the 2's complement to the minuend.
- If there is no carry generated in the operation, then 2's complement of the result is taken which will be negative.
- If there is a carry bit 1, then this carry is ignored and the result is taken which will be positive.

1. When no Carry bit:

Solve 11001 – 11100

In decimal form: $(25)_{10}$ **(Minuend)** – $(28)_{10}$ **(Subtrahend)** = $(-3)_{10}$

Following the steps of the above algorithm, 2's complement of subtrahend 11100 is taken to be (00011 + 1) = 00100. Then this 2's complement is added to the minuend, **11001 + 00100 =11101**. As there is no carry generated in the output, so 2's complement of the above result is taken to be the final output, which is **00011.** A final result is a negative number as verified with the decimal subtraction the result is (-3).

2. When Carry bit 1:

Solve 10101 – 00101

In decimal form: $(21)_{10}$ **(Minuend)** – $(5)_{10}$ **(Subtrahend)** = $(16)_{10}$

The first step is to take 2's complement of the subtrahend 00101 which is 11011. The second step is to add this 2's complement to the minuend, **10101 + 11011 =1 10000**. Here, we can see a carry is generated, so according to the algorithm, this carry is dropped and the final result will be 10000 which is a positive number. For verification, the decimal subtraction also gives the result as (+16).

Addition using 2's complement method

The addition of unsigned binary numbers is similar to that of the simple mathematical addition. The difference occurs when the addition is performed

on the two signed integers using 2’s complement method. To understand better, let’s take the most prominent cases of the addition operation and study them with the help of examples.

1. ***Addition of positive and negative number:***

- When a positive number has a greater magnitude

  If the magnitude of a positive number is greater, then the addition can be carried out by simply taking 2’s complement of the negative number. The carry bit is ignored and the result will be a positive number. For example, if we have to add 1110 (14_{10}) and -1101 (-13_{10}), then 2’s complement of 1101 will be 0011 which is added to 1110. The result will be 1 0001, the carry bit 1 is ignored and it is a positive number +0001 (1_{10}).

- When a negative number has a greater magnitude

  If the magnitude of a negative number is greater, then the addition can be carried out by simply taking 2’s complement of the negative number. As there is no carry generated in this case, then 2’s complement of the result is taken that comes out to be a negative number. For example, if we have to add 01010 (10_{10}) and -01100 (-12_{10}), then 2’s complement of 01100 will be 10100, that is added to 01010. The result will be 11110. Since there is no carry generated, 2’s complement of the result is taken, i.e. 00010 and it is a negative number -00010 (-2_{10}).

1. ***Addition of positive and negative number:***

   If both the given numbers are negative, then 2’s complement of both the numbers is taken and added. The carry generated is dropped and 2’s complement of the result is taken again, the answer is obtained and it is a negative number. Alternatively, both the binary numbers can be added directly, obtaining the result as a negative number. For example, if we have to add -01010 (-10_{10}) and -00101 (-5_{10}), then 2’s complement of both numbers is taken and added, i.e. 10110+11011 =1 10001. As there is a carry bit generated, it is dropped and 2’s complement of the result 10001 is taken again. The result will be 01111 and it is a negative number -01111 (-15_{10}).

3.4 Multiplication of Binary Numbers

Multiplication of binary numbers is a complex operation. Different methods are used for the multiplication of unsigned and signed integers. The multiplication of an unsigned number is just similar to the decimal multiplication, following the basic rules: 0 x 0=0, 0 x 1= 0, 1 x 0=0, and 1 x 1=1. For example, if we have to multiply 1011 (11_{10}) by 1001(9_{10}), then

```
    1011
   X1001
 -------
 0001011
 000000x
 00000xx
 1011xxx
 -------
 1100011 = (99₁₀)
```

For signed integers, the simple multiplication is not successful as the most significant bit or sign bit represents the sign of the number, so it becomes difficult to perform the multiplication operation, even using the 2's complement method. The method that can be used for multiplication of signed integers is: First both the given numbers are converted to positive numbers and then multiplied with each other. 2's complement is taken for the result. But, to be more accurate and to avoid the final 2's complement, another method is followed i.e. **Booth's Algorithm.**

3.4.1 Booth's Algorithm

Booth's algorithm is a fast and efficient technique for multiplication of signed binary numbers. It is used for both positive and negative numbers. *Andrew D. Booth* introduced Booth's algorithm to make the multiplication operation easy. In the normal multiplication process, when we start the operation with the right end bit and multiply all the digits of the multiplicand with it; the product obtained from this first step is called a *partial product.* Then, after shifting a place, partial product from the next digit is obtained and the process of shifting and multiplying continues till the left-end digit is multiplied by the multiplicand. This process is followed by the addition of all the partial products

to get the final product of the multiplication process. In Booth's algorithm, the shifting is termed as *Arithmetic Right- Shift,* in which all the bits are shifted to one bit right, and the least significant bit is dropped, while the most significant bit remains unchanged. Since, according to the signed magnitude representation of binary numbers, the most significant bit represents the sign of the number; so, the arithmetic right shift gives an advantage of not changing the sign bit, resulting in ease in handling the signed binary numbers.

Booth's algorithm uses certain registers to accomplish the process, the arrangement is shown in figure 3.2. The multiplicand is loaded in the register MD and the multiplier in MR. Both MD and MR are n-bit registers. PD is an n-bit register to store the final product. CR refers to the n-bit counter ranging from n to 0. The operation is considered to be completed when the counter displays zero value. MX is a 1-bit register used in the shifting process.

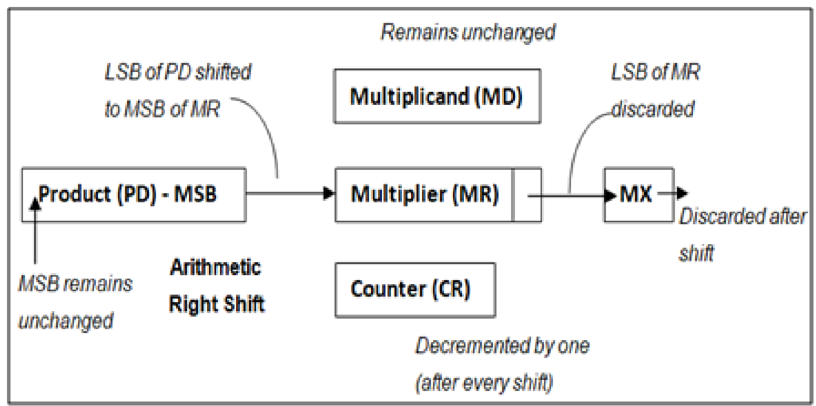


Figure 3.2 Registers involved in Booth's algorithm

In Booth's algorithm, the following conditions are being processed before every shifting. These conditions are to be checked for a combination of two bits, the least significant bit (LSB) of the multiplier (MR) and 1-bit MX. If both bits are the same, i.e. **00** or **11** then no action is implemented and directly proceeds for the shift right process. If the bits are in pattern **10,** then the content of multiplicand (MD) is subtracted from present content of product PD which is stored in PD; while if the bits display **01,** the addition of MD and PD is implemented and the result is stored in PD. Borrowing and carry generation in

the subtraction and addition operation is neglected. Each time after addition and subtraction, the content of PD is overwritten with the new result. The LSB of MR is discarded each time from MX after shifting.

MR (LSB)	MX	Remarks
0	0	No action required
1	1	No action required
1	0	Subtract MD from PD, giving results in PD. (PD ← PD – MD)
0	1	Add MD with PD, giving results in PD. (PD ← PD + MD)

Figure 3.3 Conditions for booth's algorithm before every shift

Booth's algorithm is accomplished using the following steps. Figure 3.4 shows the flowchart of Booth's algorithm.

1. The multiplicand is loaded in MD and the multiplier is loaded in MR. For the signed (negative) number, 2's complement method is applied.
2. Counter CR is initialized with the number of bits involved.
3. Clear PD and MX.
4. Check the LSB of MR and MX in combination. If the obtained pattern is 00 or 11, then proceed for step-5, while if the pattern is 10, then PD= PD- MD and if the pattern is 01, then PD= PD + MD.
5. Arithmetic Right shift operation is performed with PD, MR, and MX. LSB of PD goes to MSB of MR and LSB of MR goes to MX and the old content of MX is discarded.
6. Counter CR is decremented by 1. If CR is not zero then again go to step-4.
7. Finally, the result is available in PD (upper part) and MR (lower part).

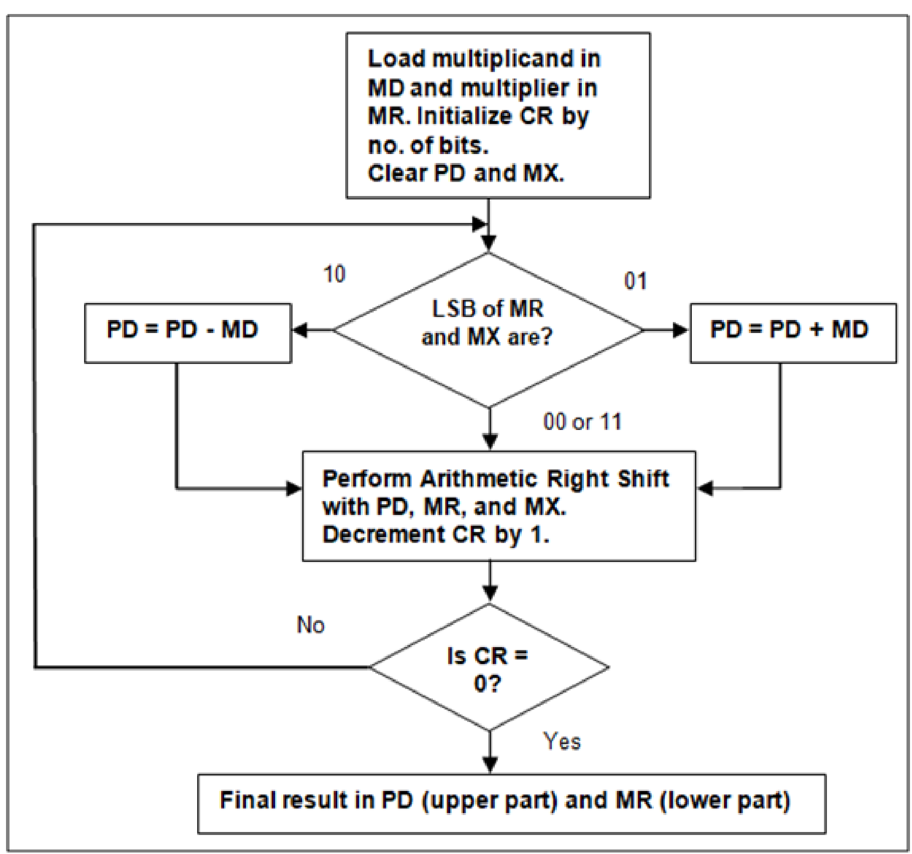


Figure 3.4 Flowchart for Booth's Algorithm

Example of Booth's algorithm

Q- Multiply -6 (0110) by 7 (0101) using the booth's algorithm.

Solution- Given, multiplicand, MD= -6 (0110) with 2's complement 1010, multiplier, MR= 7 (0111). Counter CR is initialized with 4. Now, the conditions for Booth's algorithm are checked, as shown in the table below:

Steps	Product, PD	Multiplier, MR	MX	Counter, CR	Remarks
Step-1	0000 0110 0011	0111 0111 0011	0 0 1	4 4 3	PD= PD-MD. (0000- 1010= 0110 by 2's complement method) CR= CR-1 Shifting

Step-2	0011 0001	0011 1001	1 1	3 2	No action, CR= CR-1 Shifting
Step-3	0001 0000	1001 1100	1 1	2 1	No action, CR= CR-1 Shifting
Step-4	0000 1010 1101	1100 1100 0110	1 1 0	1 1 0	PD= PD+MD (0000+1010 = 1010) CR= CR-1 Shifting

As CR=0, so the process will stop and the final product will be in PD (upper part) and MR (lower part)

Final product= PD MD = 1101 0110

Now taking the 2's complement again, as the given multiplicand is a signed integer. The MSB will not be complemented while taking the 2's complement as it indicates the sign of the binary number.

Now, Final product= 10101010= $(-42)_{10}$, where MSB=1 indicates that the product is negative.

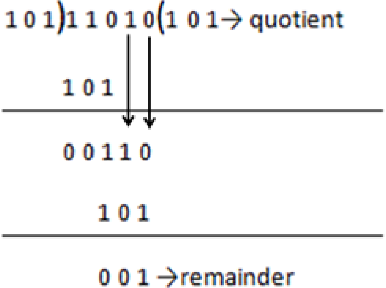
3.5 Division of Binary Numbers

The division of binary numbers is somewhat similar to the division of decimal numbers, following the long division method. There is a repetitive shifting and addition or subtraction involved in the operation. In unsigned binary division, firstly, the bits of the dividend are checked from left to right which should be greater than or equal to the divisor, showing that the divisor is able to divide the given dividend.

For example, we have to divide $(26)_{10} = (11010)_2$ by $(5)_{10} = (101)_2$

In the first step of the division operation, the quotient is kept as 1, as the divisor multiplied by the quotient will give us the partial dividend that is to be subtracted from the dividend i.e. 101 is subtracted from 110, giving the partial

remainder as 001. Now, checking the rule that the dividend should be greater or equal to the divisor, 0 is inserted in the quotient and now the partial dividend becomes 110. Again divisor 101 is subtracted from 110 giving us the final remainder 001 and final quotient 101.



For the binary division of signed numbers, 2's complement method is incorporated. The signed numbers are converted to unsigned integers with the help of 2's complement method and then the long division method is followed for the division operation.

3.6 Floating-point Representation

To represent very small or very large numbers, a method other than fixed-point representation is required i.e. Floating- point notation. With the help of fixed-point representation, the smallest range of positive and negative numbers near zero (0) can be denoted. But, fixed-point representation has a limitation for representing very large and very small fractions of numbers. In decimal number system, a very small or large number can be represented in the form of exponents to the base, for example, a number 0.156 can be represented as 15.6×10^{-2}, where 10^{-2} shows the exponent power of -2 to the base 10, 15.6 is called a fixed-point mantissa. As the given number can be denoted in other forms too by changing the exponent value, it is known as floating-point representation.

The binary numbers can be represented in floating-point notation, in three parts: a **sign** bit (0- plus or 1- minus) the **exponent** part, and a **fractional** part (also known as **Mantissa**). Although, the term mantissa also denotes the fractional part of the logarithm and it is different from the fractional part of the

floating-point notation. According to IEEE, this fractional part is termed as **significand**. The radix point is assumed to be the most significant bit of the binary number. For example, the binary number +1100.11 will be represented with a fraction of 8 bit and exponent 0 in floating-point notation as 0.1100110 x 2^{100}, where the fraction is 01100110 and exponent is 000100. Here, the MSB is 0 which denotes that it is a positive number.

3.7 Floating-Point Arithmetic and Unit Operations

The floating-point arithmetic is somewhat different from the decimal and binary arithmetic. There are four common arithmetic operations, addition, subtraction, multiplication, and division. It is essential to check the exponent values of the operands while performing any floating-point arithmetic operation. In the case of addition and subtraction, the exponent value of both the operands should be the same. The radix point is shifted accordingly to make the exponent values the same. One of the following conditions can be obtained from the floating-point operation:

- **Exponent overflow:** It is possible that the positive exponent value exceeds the maximum possible value.
- **Exponent underflow:** This condition may occur when the value of a negative exponent is less than the minimum possible value.
- **Significand overflow:** This condition can persuade when the addition of two significands of the same sign can result in a carry out of the most significant bit (MSB).
- **Significand Underflow:** It is possible while aligning significands, the digits may flow off from the right end of the significand.

3.7.1 Floating-point Addition and Subtraction

The floating-point addition and subtraction operations are more complex than the multiplication and division operations. This is due to one reason, *alignment* of significands. The addition and subtraction algorithm of floating-point numbers takes place in four steps:

1. **Checking for zeroes:** In the process of subtraction, the sign of the subtrahend is changed and if any of the operands is 0, the other is considered to be the result.
2. **Align the significands:** Alignment of the significands is done by shifting the larger number to the left (decreasing the exponent) or shifting the smaller number to the right (increasing the exponent).
3. **Addition or subtraction:** Performing the desired operation on the aligned significands.
4. **Normalization of the result:** The result is normalized by shifting the significand bits to the left until the most significant bit comes to be non-zero.

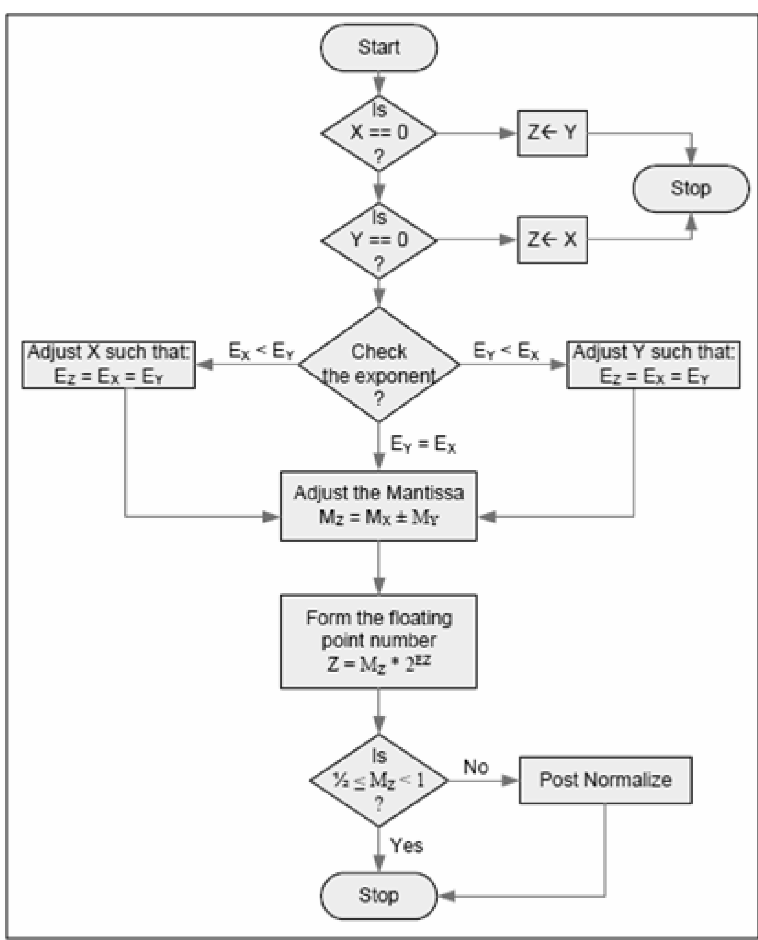


Figure 3.5 A Flow- chart for floating-point addition and subtraction

(**Source-**http://www.ioenotes.edu.np/media/notes/computer-organization-and-architecture-coa/Chapter5-Computer-Arithmetic.pdf)

3.7.2 Floating-point Multiplication

The floating-point multiplication is carried out in the following four steps:

1. Check for zeros
2. Add the exponents
3. Multiply the significand
4. Normalization of result

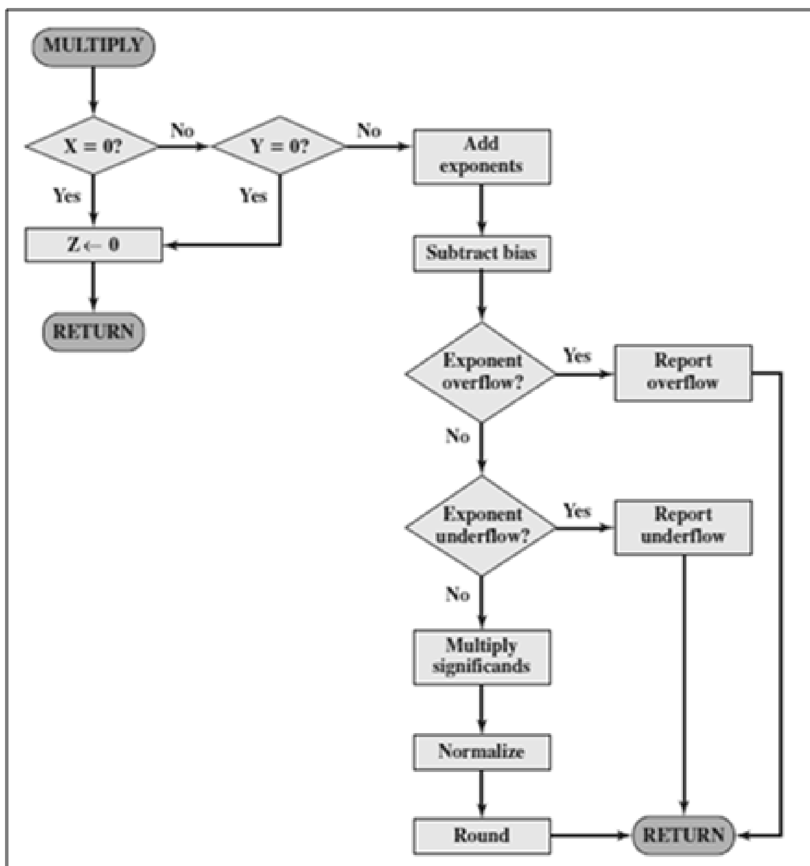


Figure 3.6 Flow- chart for floating-point Multiplication

(**Source-** Computer Organization and Architecture, Ninth Edition, William Stallings, Chapter-10, Page no. 353)

3.7.3 Floating-point Division

The floating-point division is carried out in the following five steps:

1. Check for zeros
2. Align the dividend
3. Subtraction of exponent

4. Divide the significand
5. Normalization of result

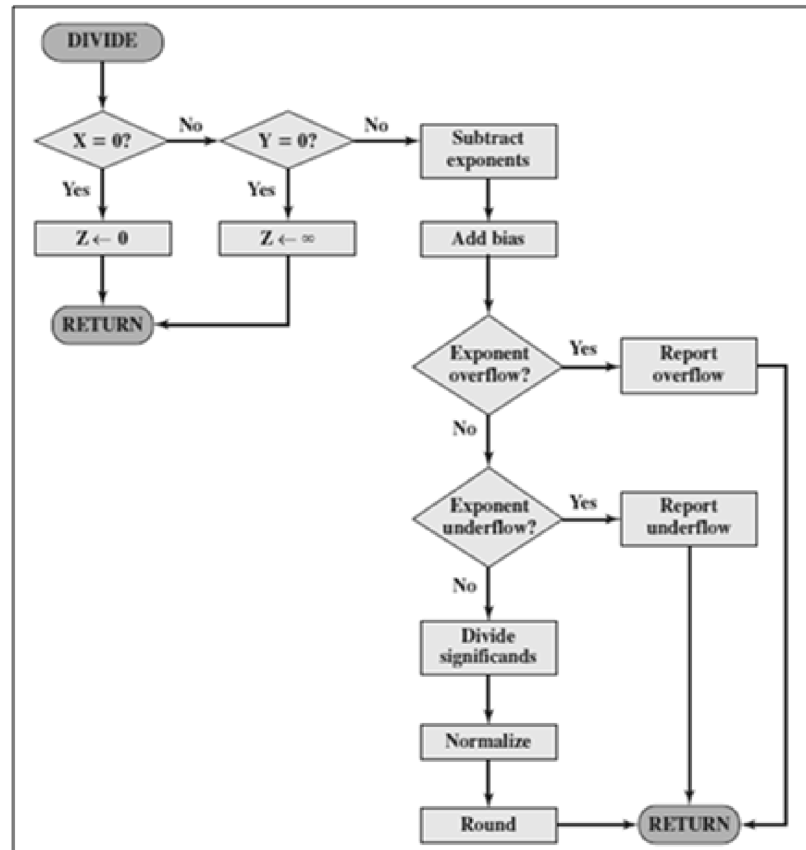


Figure 3.7 Flow- chart for floating-point Division

(**Source-** Computer Organization and Architecture, Ninth Edition, William Stallings, Chapter-10, Page no. 354)

3.8 Binary Codes and Error Detection Codes

Besides Fixed point and Floating point data representation, the digital computers employ certain binary codes for specified operations and applications. As we know that the digital systems are based on discretization of data but there are certain systems that follow a continuous data output approach. For such systems, there is a need to convert the continuous form of data to digital form with the help of an analog to digital converter. The data obtained from such conversion is in the form of codes.

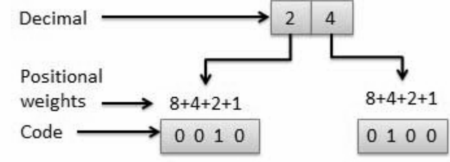
The digital data represented, stored and transmitted as a group of binary bits, is called ***binary code.*** The binary code can be represented by the number as well as the alphanumeric letter. Binary codes are suitable for computer applications, digital communications, 0 & 1 are being used, implementation becomes easy. Before discussing the types of binary codes, let's introduce binary decimal code (BCD) codes.

In a ***binary decimal code (BCD)***, each decimal digit is represented by a 4-bit binary number. BCD is a way to express each of the decimal digits with a binary code. In the BCD, with four bits we can represent sixteen numbers (0000 to 1111). But in BCD code only the first ten of these are used (0000 to 1001). The remaining six code combinations i.e. 1010 to 1111 are invalid in BCD.

Decimal	0	1	2	3	4	5	6	7	8	9
BCD	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

The binary codes are broadly classified as *weighted codes, non-weighted codes, reflective codes, sequential codes, alphanumeric codes, error detecting and correcting codes.*

- ***Weighted Codes:*** In weighted codes, each digit is assigned a specific weight according to its position. For example, in 8421 BCD code, 1001 the weights of 1, 0, 0, 1 (from left to right) are 8, 4, 2 and 1 respectively. The codes ***8421 BCD, 2421 BCD, 5211 BCD*** are all weighted codes.



- ***Non-Weighted Codes:*** The non-weighted codes are not positionally weighted. In other words, each digit position within the number is not assigned a fixed value (or weight). ***Excess-3*** and ***gray code*** are non-weighted codes.

Excess-3 code: The Excess-3 code is also called XS-3 code. It is a non-weighted code used to express decimal numbers. The Excess-3 code words are derived from the 8421 BCD code words adding (0011)2 or (3)10 to each code word in 8421. The excess-3 codes are obtained as follows:

Decimal Number ⟶ 8421 BCD —Add 0011⟶ Excess-3

Gray Code: It is the non-weighted code and non-arithmetic code. That means there are no specific weights assigned to the bit position. It has a very special feature that only one bit will change each time the decimal number is incremented as shown below. As only one bit changes at a time, the gray code is called a ***unit distance code***. The gray code is a cyclic code and cannot be used for arithmetic operations.

Decimal	BCD	Gray
0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 0 1 0
4	0 1 0 0	0 1 1 0
5	0 1 0 1	0 1 1 1
6	0 1 1 0	0 1 0 1
7	0 1 1 1	0 1 0 0
8	1 0 0 0	1 1 0 0
9	1 0 0 1	1 1 0 1

- ***Reflective codes:*** A code is reflective when the code is self complementing. In other words, when the code for 9 is the complement of the code for 0, 8 for 1, 7 for 2, 6 for 3 and 5 for 4. ***2421 BCD, 5421 BCD and Excess-3*** code are reflective codes.
- ***Sequential codes:*** In sequential codes, each succeeding 'code is one binary number greater than its preceding code. This property helps in manipulation of data. ***8421 BCD and Excess-3*** are sequential codes.
- ***Alphanumeric codes:*** Codes used to represent numbers, alphabetic characters, symbols and various instructions necessary for conveying intelligible information. ***ASCII*** (American Standard Code for Information

Interchange), ***EBCDIC*** (Extended Binary Coded Decimal Interchange Code)***, UNICODE*** are the most-commonly used alphanumeric codes. ASCII code is a 7-bit code whereas EBCDIC is an 8-bit code. ASCII code is more commonly used worldwide while EBCDIC is used primarily in large IBM computers.

- ***Error detecting and correcting codes:*** Codes which allow error detection and correction are called error detecting and correcting codes. ***Parity and Hamming code*** are the mostly commonly used error detecting and correcting codes.

Error detection and correction code plays an important role in the transmission of data from one source to another. The noise also gets added into the data when it transmits from one system to another, which causes errors in the received binary data at other systems. The bits of the data may change (either 0 to 1 or 1 to 0) during transmission. It is impossible to avoid the interference of noise, but it is possible to get back the original data. For this purpose, we first need to detect whether an error **z** is present or not using error detection codes. If the error is present in the code, then we will correct it with the help of error correction codes. Parity check, Checksum, and CRC are the error detection techniques.

The ***error detection codes*** are the code used for detecting the error in the received data ***bitstream***. Error detecting codes encode the message before sending it over the noisy channels. The encoding scheme is performed in such a way that the decoder at the receiving end can find the errors easily in the receiving data with a higher chance of success.

Parity Code: In parity code, we add one parity bit either to the right of the LSB (least significant bit) or left to the MSB (most significant bit) to the original bitstream. On the basis of the type of parity being chosen, two types of parity codes are possible, i.e., ***even parity*** code and o***dd parity*** code. These codes are used when we use message backward error correction techniques for reliable data transmission. A ***feedback*** message is sent by the receiver to inform the sender whether the message is received without any error or not at

the receiver side. If the message contains errors, the sender retransmits the message.

In error detection codes, in fixed-size blocks of bits, the message is contained. In this, the redundant bits are added for correcting and detecting errors. These codes involve checking for errors. No matter how many error bits are there and the type of error.

Error correction codes are generated by using the specific algorithm used for removing and detecting errors from the message transmitted over the noisy channels. The error-correcting codes find the correct number of corrupted bits and their positions in the message. There are two types of ECCs (Error Correction Codes), which are as follows.

a) **Block codes:** In block codes, in fixed-size blocks of bits, the message is contained. In this, the redundant bits are added for correcting and detecting errors.

b) **Convolutional codes:** The message consists of data streams of random length, and parity symbols are generated by the sliding application of the Boolean function to the data stream.

The hamming code technique is used for error correction.

Hamming Code: Hamming code is an example of a block code. The two simultaneous bit errors are detected, and single-bit errors are corrected by this code. In the hamming coding mechanism, the sender encodes the message by adding the unessential bits in the data. These bits are added to the specific position in the message because they are the extra bits for correction.

3.9 Summary

- The computer system is dependent on four types of number systems, binary, decimal, octal, and hexadecimal. With the help of conversion techniques, the computer converts the input in binary form and then performs the desired operation to give the output.

- The arithmetic operations are different for signed and unsigned numbers. Two's complement method is the most prominent method used for unsigned number operations.
- Booth's algorithm gives the most accurate result for the multiplication of unsigned numbers.
- To represent very small or very large numbers, a method other than fixed-point representation is required i.e. Floating- point notation. The binary numbers can be represented in floating-point notation, in three parts: a sign bit, the exponent part, and a fractional part (also known as Mantissa or significand).
- The digital data represented, stored and transmitted as a group of binary bits, is called ***binary code.***
- The binary codes are broadly classified as *weighted codes, non-weighted codes, reflective codes, sequential codes, alphanumeric codes, error detecting and correcting codes.*

3.9 Key Terms

- **Most Significant Bit (MSB):** In signed magnitude representation, the leftmost bit represents the sign of the binary number. This bit is known as the most significant bit.
- **Least Significant Bit (LSB):** The rightmost bit of the binary number is known as the least significant bit.
- **Signed Magnitude Representation:** It is the method of representing binary numbers with their sign (positive and negative). The MSB denotes the sign of the binary number. MSB- 0 means the number is positive and MSB- 1 means the number is negative.
- **Mantissa:** The fractional part of the logarithm is called the mantissa. In a floating-point representation system, the fractional part is termed as mantissa or significand, though both the fractional parts are different from each other.

3.10 Check Your Progress

Q1) What is the main disadvantage of using signed magnitude representation?

Q2) Explain the two's complement method with an example.

Q3) Discuss the binary to decimal and decimal to binary conversion techniques with a suitable example.

Q4) Solve the following:

a) $(0010011)_2 = (?)_{10}$ b) $(30)_2 = (?)_{16}$ c) $(65)_{10} = (?)_2$ d) $(1AE)_{16} = (?)_2$

Q5) What is the Arithmetic Right Shift operation? Explain.

Q6) What are the steps involved in Booth's Algorithm? Also, draw the flow-chart for the same.

Q7) Solve:

a) Add 10110 and 110110

b) Multiply (1011) with (1100)

c) Subtract 110111 from 01011000

d) Multiply (1010) with (-1101)

References:

Computer Architecture and Organization, Subrata Ghoshal, Pearson Publication.

Computer Organization and Architecture, *9th edition*, William Stallings, Pearson Publication.

Computer System Architecture, M. Morris Mano.

http://www.pvpsiddhartha.ac.in/dep_it/lecturenotes/CSA/unit-4.pdf

https://www.tutorialspoint.com/two-s-complement

https://www.geeksforgeeks.org/computer-organization-booths-algorithm/

https://www.electrical4u.com/binary-division/

http://www.ioenotes.edu.np/media/notes/computer-organization-and-architecture-coa/Chapter5-Computer-Arithmetic.pdf

Unit 4 – Register Transfer and Micro-operations

Structure

4.0 Introduction
4.1 Unit Objectives
4.2 Register Transfer Language
4.3 Register Transfer
4.4 Bus and Memory Transfers
4.5 Logic micro-operations
4.6 Shift micro-operations
4.7 Arithmetic Logic Shift Unit
4.8 Summary
4.9 Key Terms
4.10 Check Your Progress

4.0 Introduction

A digital system can be defined as an interconnection of digital hardware modules that execute a specific information-processing task. Digital systems pursue a variation in size and complexity from a few integrated circuits to a complex of interconnected and interacting digital computers. Digital system design invariably uses a modular approach. The modules are constructed from such digital components as registers, decoders, arithmetic elements, and control logic. The various mod-ules are interconnected with common data and control paths to form a digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the data stored in them. The operations executed on data stored in registers are called ***microoperations***.

A microoperation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the

previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, clear, and load. A bidirectional shift register is capable of performing the shift right and shift left microoperations.

The specifications for the internal hardware organization of a digital computer are:

- The set of registers it contains and their function.
- The sequence of microoperations performed on the binary information stored in the registers.
- The control that initiates the sequence of microoperations.

4.1 Unit Objectives

This unit will help the reader to gain knowledge about:

- Register Transfer and Register Transfer Language
- Bus and Memory Transfer Operations
- Microoperations and Types of microoperations
- Arithmetic Logic Shift Unit

4.2 Register Transfer Language

The symbolic notation used to describe the microoperation transfers among registers is called a ***register transfer language (RTL)***. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. The use of ***symbols*** instead of a ***narrative explanation*** provides an organized and concise manner for listing the micro-operation sequences in registers and the control functions that initiate them.

A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module. It is a convenient tool for describing the internal organization of digital computers in a concise and precise manner. It can also be used to facilitate the design process of digital systems.

The register transfer language adopted here is believed to be as simple as possible. We will proceed to define symbols for various types of microoperations, and at the same time, describe associated hardware that can implement the stated microoperations. The symbolic designation is used in this unit to specify the register transfers, the microoperations, and the control functions that describe the internal hardware organization of digital computers. Other symbology in use can easily be learned once this language has become familiar, for most of the differences between register transfer languages consist of variations in detail rather than in overall purpose.

4.3 Register Transfer

Most of the computer registers are assigned by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register holding an address for the memory unit is usually called a *memory address register* and is designated by the name *MAR*. Other designations for registers are *PC* (*program counter*), *IR* (*instruction register)*, and *R1* (*processor register*). The individual flip-flops in an n-bit register are numbered in se-quence from 0 through n-1, starting from 0 in the rightmost position and increasing the numbers toward the left.

Figure 4.1 shows the representation of registers in the computer system. The most common way to represent a register is by a rectangular box with the name of the register inside, as in figure 4.1 (a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H (for high byte). The name of the 16-bit register is PC. The symbol PC (0-7) or PC (L) refers to the low-order byte and PC (S-15) or PC (H) to the high-order byte.

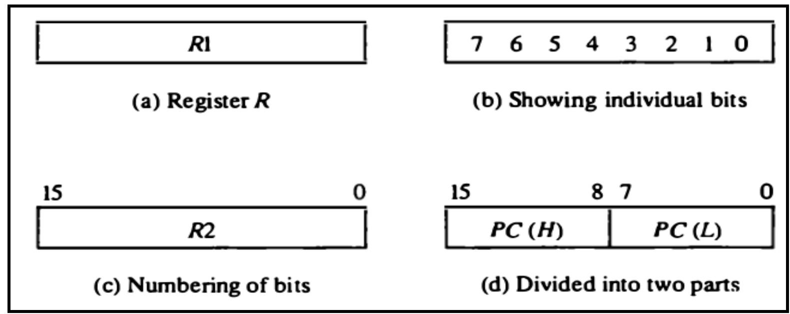


Figure 4.1 Block diagram of Register

(Source- Computer System Architecture, Morris Mano third edition**)**

Information transfer from one register to another is designated in sym-bolic form by means of a replacement operator. The statement $R2 \leftarrow R1$ denotes a transfer of the content of register R1 into register R2. It designates a replacement of the content of R2 by the content of R1. By definition, the content of the source register R1 does not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Normally, we want the transfer to occur only under a predetermined control condition. This can be shown by means of an if-then statement.

$$\text{If } (P = 1) \text{ then } (R2 \leftarrow R1)$$

where P is a control signal generated in the control section.

Control Function: A control function is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

$$P: R2 \leftarrow R1$$

The control condition is terminated with a ***colon***. It symbolizes the requirement that the transfer operation is executed by the hardware only if P = 1.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 4.2 shows the block dia-gram that depicts the transfer from R1 to R2. The n outputs of register R1 are connected to the n inputs of register R2. The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual

number when the length of the register is known. Register R2 has a load input that is activated by the control variable P. It is assumed that the control variable is synchronized with the same clock as the one applied to the register.

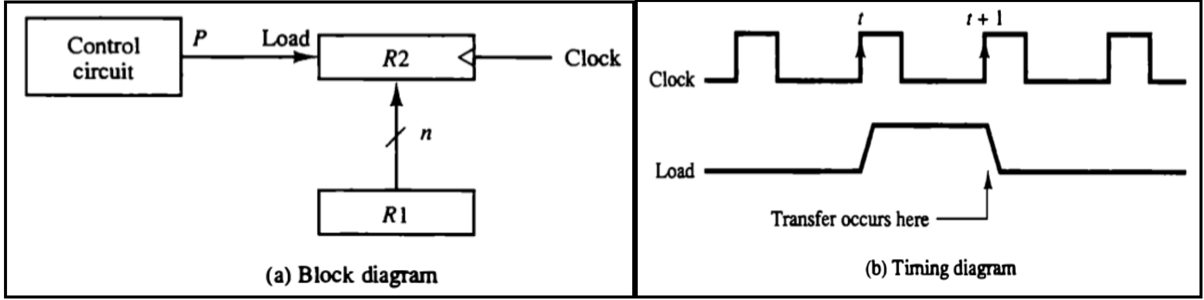


Figure 4.2 Transfer from R1 to R2 when P = 1

(Source- Computer System Architecture, Morris Mano third edition**)**

As shown in the timing diagram, P is activated in the control section by the rising edge of a clock pulse at time t. The next positive transition of the clock at time t + 1 finds the load input active and the data inputs of R2 are then loaded into the register in parallel. P may go back to 0 at time t + 1; otherwise, the transfer will occur with every clock pulse transition while P remains active.

Table 4.1 lists the basic symbols of the register transfer notation. Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time.

Table 4.1 Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and umerals)	Denote a Register	MAR, R2
Parenthesis ***( )***	Denote a part of the register	R2 (0-7), R2 (L)
Arrow ←	Denotes transfer of Information	R2 ← R1

Comma ,	Separates two microoperations	R2 ← R1, R1 ← R2

4.4 Bus and Memory Transfers

There are many registers in a typical digital computer. These registers are interconnected using certain paths to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple-register configuration is a ***common bus system***. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer. Different ways of constructing a Common Bus System

- Using Multiplexers
- Using Tri-state Buffers

Common bus system with Multiplexers

Figure 4.3 represents a common bus system of a multiplexer for four registers. Multiplexers are one of the most prominent ways of constructing a common bus system. The multiplexers select the source register whose binary information is then placed on the bus. In figure 4.3, each register has four bits, numbered 0 through 3. The bus consists of four 4 x 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, S_1 and S_0.

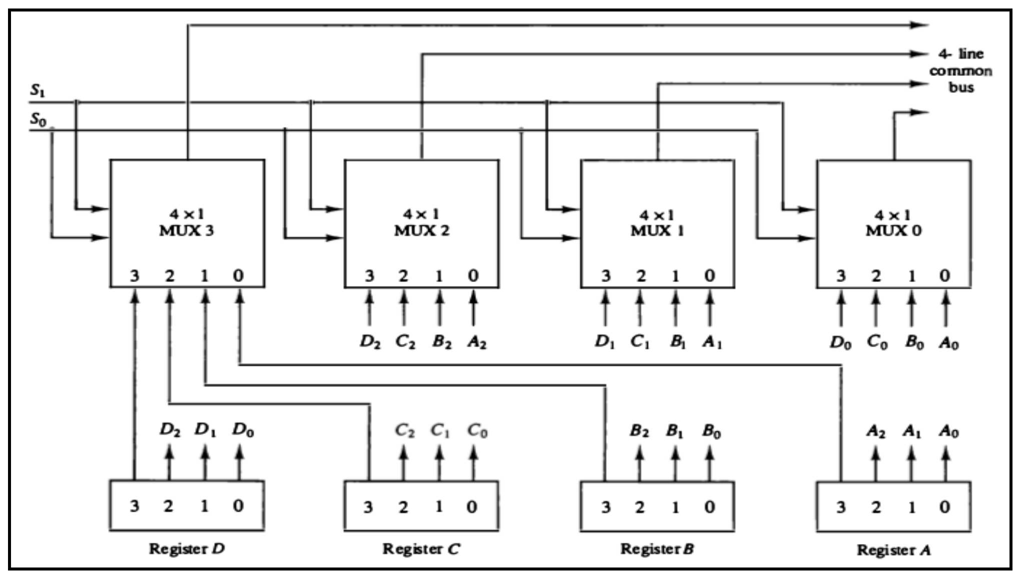


Figure 4.3 Bus System using multiplexers for four registers

(**Source-** Computer System Architecture, Morris Mano third edition)

The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits. The two selection lines S_1 and S_0 are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus. When S_1S_0 = 00, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus. This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers. Similarly, register B is selected if S_1S_0 = 01, and so on. The four possible binary values of the selection lines are:

S_1	S_0	Register Selected
0	0	A
0	1	B

1	0	C
1	1	D

We can generalize the above concept. A bus will multiplex ***k*** registers of ***n*** bits each to produce an n-line common bus. The number of multiplexers needed to construct the bus is equal to n, the number of bits in each register. The size of each multi-plexer must be ***k x 1*** since it multiplexes k data lines.

To transfer information from a bus to one of many destination registers, connect the bus lines to the inputs of all destination registers and activate the load control of the particular destina-tion register selected. The symbolic statement for a bus transfer may mention the bus or its presence may be implied in the statement. When the bus is included in the statement, the register transfer is symbolized as follows:

$$BUS \leftarrow C, R1 \leftarrow BUS$$

The content of register C is placed on the bus, and the content of the bus is loaded into register R 1 by activating its load control input. If the bus is known to exist in the system, it may be convenient just to show the direct transfer.

$$R1 \leftarrow C$$

Common bus system with Three-State Bus Buffers

A bus system can be constructed with three-state gates instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a ***high-impedance state***. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic signif-icance. Three-state gates may perform any conventional logic, such as AND or NAND. Figure 4.4 (a) depicts the graphic symbol of a three-state buffer gate. It is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any

conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input.

The high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.

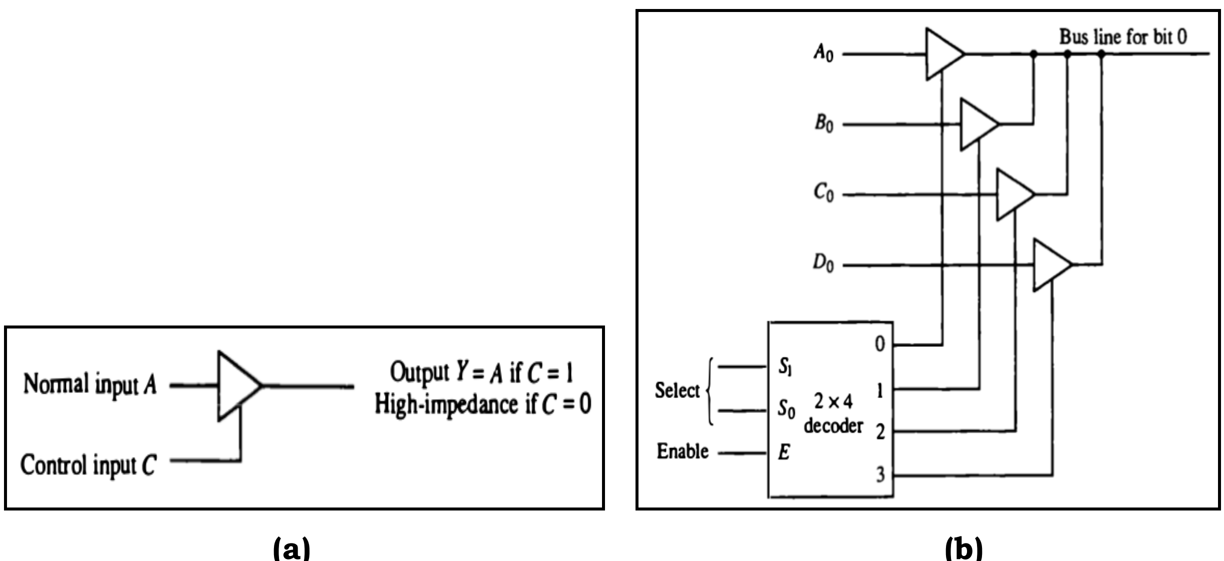


Figure 4.4 (a) Graphic symbols for three-state buffer (b) Bus line with three state-buffers.

(Source- Computer System Architecture, Morris Mano third edition**)**

In figure 4.4 (b), the outputs of four buffers are connected together to form a single bus line. The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high- impedance state.

To construct a common bus for four registers of ***n*** bits each using three-state buffers, we need n circuits with four buffers in each as shown in figure 4.5. Each group of four buffers receives one significant bit from the four registers.

Each common output produces one of the lines for the common bus for a total of ***n*** lines. Only one decoder is necessary to select between the four registers.

Memory Transfer

The transfer of information from a memory word to the outside environment is called a ***read operation***. The transfer of new information to be stored into the memory is called a ***write operation***. A memory word will be symbolized by the letter M. The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of M when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter M.

Consider a memory unit that receives the address from a register, called the ***address register*** (AR). The data are transferred to another register, called the ***data register*** (DR).

The read operation can be stated as follows:

$$\text{Read: } DR \leftarrow M\,[AR]$$

This results in a transfer of information into DR from the memory word M selected by the address in AR.

The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register R1, then:

$$R3 \leftarrow R1 + \underline{R2} + 1$$

In the above statement, $\underline{R2}$ is the symbol for 1's complement of R2. Adding 1 to the 1's complement produces the 2' s complement. Adding the contents of R1 to the 2' s complement of R2 is equivalent to R1 - R2.

Table 4.2 Arithmetic Microoperations

Symbolic Designation	**Description**
R3← R1 + R2	Contents of R1 plus R2 transferred to R3
R3← R1 - R2	Contents of R1 minus R2 transferred to R3

R2 ←$\overline{R2}$	Complement the contents of R2 (1's complement)
R2 ←$\overline{R2}$ + 1	2's complement the contents of R2 (negate)
R3 ← R1 + $\overline{R2}$ + 1	R1 plus the 2's complement of R2 (subtraction)
R1← R1 + 1	Increment the contents of R1 by one
R1← R1 - 1	Decrement the contents of R 1 by one

Table 4.2 represents arithmetic microoperations. The increment and decrement microoperations are symbolized by plus- one and minus-one operations, respectively. These microoperations are imple-mented with a combinational circuit or with a binary up-down counter.

It should be noted that the arithmetic operations of multiply and divide are not listed in Table 4.2. These two operations are valid arithmetic operations but are not included in the basic set of microoperations. The only place where these operations can be considered as microoperations is in a digital system, where they are imple-mented by means of a combinational circuit.

- **Binary Adder:**

  A Digital circuit that forms the arithmetic sum of 2 bits and the previous carry is called ***Full Adder*** whereas the digital circuit that generates the arithmetic sum of 2 binary numbers of any length is called ***Binary Adder***. The binary adder is constructed with full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. Figure 4.6 shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The input carry to the binary adder is C_0 and the output carry is C_4. The S outputs of the full-adders generate the required sum bits. An n-bit binary adder requires n full-adders.

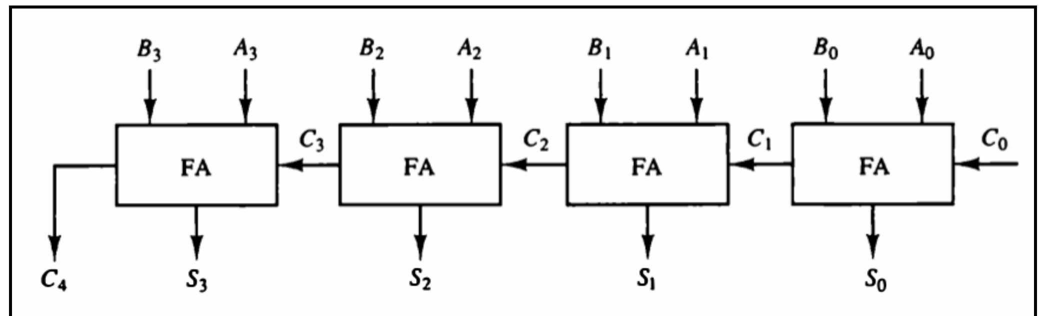


Figure 4.5 4-bit binary adder

- **Binary Adder- Subtractor:**

  The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full adder. A 4-bit adder-subtractor circuit is shown in figure 4.6. The mode input M controls the operation. When M = 0 the circuit is an adder and when M = 1 the circuit becomes a subtractor. Each exclusive-OR gate receives input M and one of the inputs of B. When M = 0, we have $B \oplus 0 = B$. The full-adders receive the value of B, the input carry is O, and the circuit performs A plus B. When M = 1, we have $B \oplus 1 = B'$ and $C_0 = 1$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs operation A plus the 2's complement of B. For unsigned numbers, this gives A - B if A ≥ B or the 2's complement of (B - A) if A < B. For signed numbers, the result is A - B provided that there is no overflow.

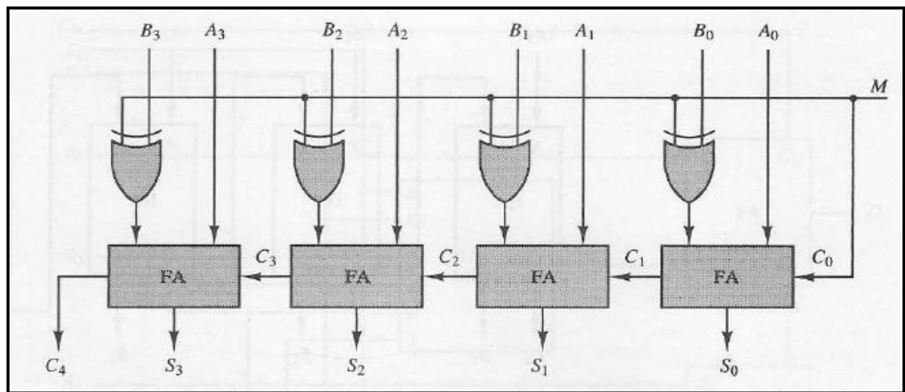


Figure 4.6 4-bit binary adder- subtractor

4.5 Logic micro-operations

Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers R1 and R2 is symbolized by the statement:

$$P: R1 \leftarrow R2 \oplus R2$$

It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable P = 1. The content of R1, after the execution of the microoperation, is equal to the bit-by-bit exclusive-OR operation on pairs of bits in R2 and previous values of R1.

There are ***16*** different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in Table 4.3. In this table, each of the 16 columns F_0 through F_{15} represents a truth table of one possible Boolean function for the two variables x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F.

Table 4.3 Truth Tables for 16 Functions of Two Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The Boolean functions listed in the first column of Table 4.4 repre-sent a relationship between two binary variables x and y. The logic micro-operations listed in the second column represent a relationship between the binary content of two registers A and B.

Table 4.4 Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

The ***hardware implementation*** of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic microoperations, most computers use only four-AND, OR, XOR (exclusive-OR), and complement- from which all others can be derived.

4.6 Shift Microoperations

Shift microoperations are used for the serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. The contents of a register can be shifted to the left or the right. At the same time that the bits are shifted, the first flip-flop receives its binary information from the serial input. During a shift-left operation, the serial input transfers a bit into the rightmost position. During a shift-right operation, the serial input transfers a bit into the leftmost position. The information transferred through the serial input determines the type of shift. There are three types of shifts: ***logical, circular, and arithmetic***.

- **Logical Shift:** A logical shift is one that transfers 0 through the serial input. We will adopt the symbols ***shl*** and ***shr*** for logical shift-left and shift-right microoperations, respectively. For example:

  $$R1 \leftarrow \text{shl } R1$$

  $$R2 \leftarrow \text{shr } R2$$

  These two microoperations specify a 1-bit shift to the left of the content of register R1 and a 1-bit shift to the right of the content of register R2. The register symbol must be the same on both sides of the arrow. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.
- **Circular Shift:** The circular shift (also known as a ***rotate operation***) circulates the bits of the register around the two ends without loss of information. This is accom-plished by connecting the serial output of the shift register to its serial input. We will use the symbols ***cil*** and ***cir*** for the circular shift left and right, respec-tively. The symbolic notation for the shift microoperations is shown in Ta-ble 4.5.

Table 4.5 Shift Microoperations

Symbolic Representation	**Description**
R ← shl R	Shift-left register R
R ← shr R	Shift-right register R
R ← cil R	Circular shift-left register R
R ← cir R	Circular shift-right register R
R ← ashl R	Arithmetic shift-left R
R ← ashr R	Arithmetic shift-right R

- **Arithmetic Shift:** An arithmetic shift is a microoperation that shifts a signed binary number to the left or right. An arithmetic shift-left

multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2's complement form.

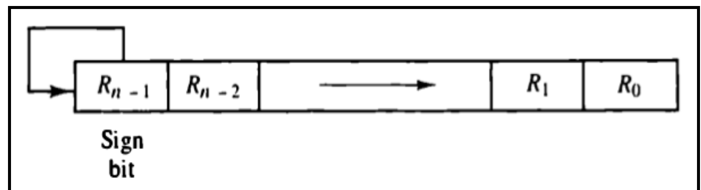


Figure 4.7 Arithmetic Shift Right

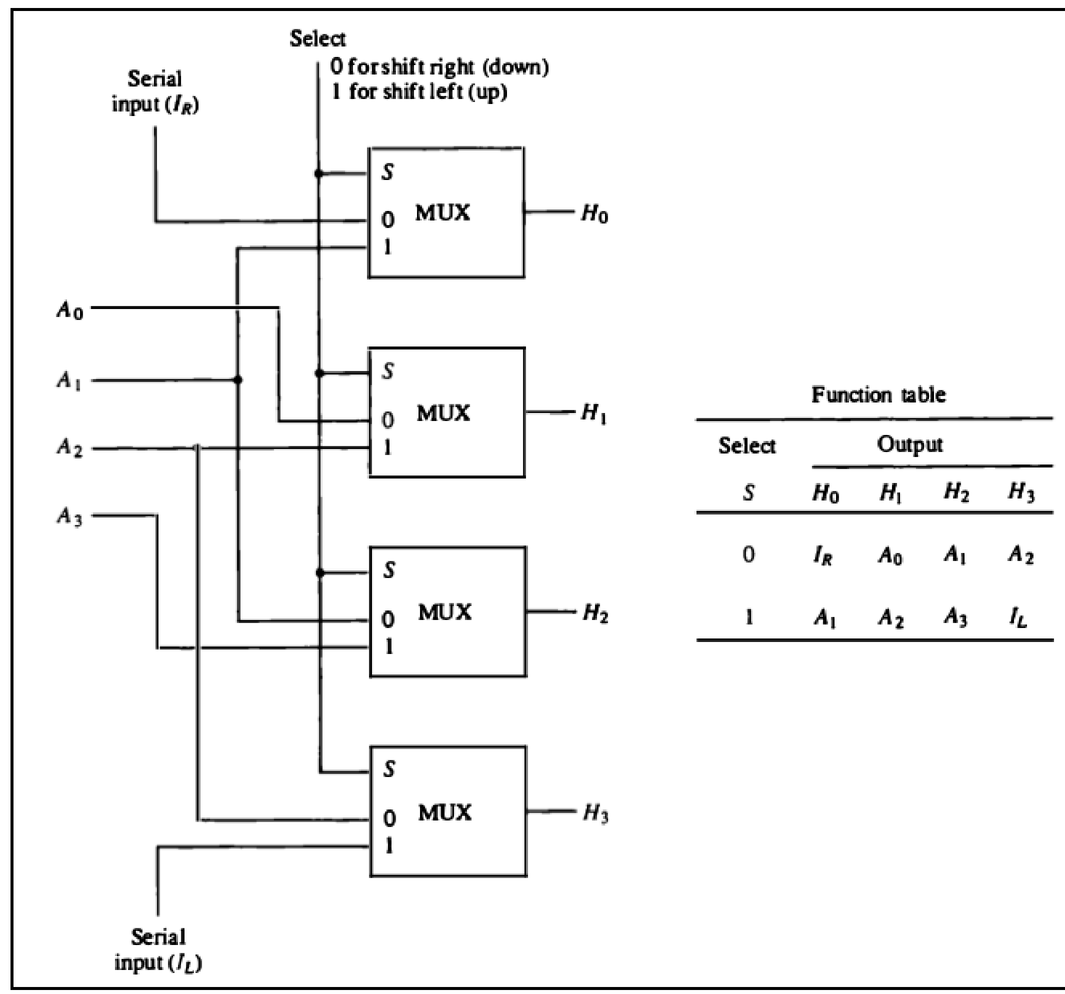
As shown in the above figure, bit R_{n-1} in the leftmost position holds the sign bit. R_{n-2} is the most significant bit of the number and R_0 is the least significant bit. The arithmetic shift-right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Thus, R_{n-1} remains the same, R_{n-2} receives the bit from R_{n-1} and so on for the other bits in the register. The bit in R_0 is lost.

The arithmetic shift-left inserts a 0 into R_0 and shifts all other bits to the left. The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2}. A sign reversal occurs if the bit in R_{n-1} changes in value after the shift. This happens if the multiplication by 2 causes an overflow. An overflow occurs after an arithmetic shift left if initially, before the shift, R_{n-1} is not equal to R_{n-2}. An overflow flip-flop V can be used to detect an arithmetic shift-left overflow. If $V_s = 0$, there is no overflow, but if $V_s = 1$, there is an overflow and a sign reversal after the shift. V_s must be transferred into the overflow flip-flop with the same clock pulse that shifts the register.

$$V_s = R_{n-1} \oplus R_{n-2}$$

For ***hardware implementation***, a combinational circuit shifter can be constructed with multiplexers as shown in figure 4.8. The 4-bit shifter has four data inputs, A_0 through A_3, and four data outputs, H_0 through H_3. There are two serial inputs, one for shift left (I_L) and the other for

shift right (I_R). When the selection input S = 0, the input data are shifted right (down in the diagram). When S = 1, the input data are shifted left (up in the diagram). The function table in figure 4.8 shows which input goes to each output after the shift. A shifter with n data inputs and outputs requires n multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.



Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Figure 4.8 4-bit Combinational Circuit Shifter

4.7 Arithmetic Logic Shift Unit

So far we have studied that individual registers perform all the microoperations directly. Besides this, the computer systems employ a number of storage registers connected to a com-mon operational unit, ***arithmetic logic unit*** (ALU). To perform a microoperation, the contents of specified registers are

placed in the inputs of the common ALU. The ALU performs an operation and the result of the operation is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period. The shift microoperations are often performed in a separate unit, but sometimes the shift unit is made part of the overall ALU.

The arithmetic, logic, and shift circuits can be combined into one ALU with common selection variables. Figure 4.9 represents one stage of an arithmetic logic shift unit. The subscript ***i*** designate a typical stage. Inputs A_1 and B_1 are applied to both the arithmetic and logic units. A particular microoperation is selected with inputs S_1 and S_0. A 4 x 1 multiplexer at the output chooses between an arithmetic output in E_i and a logic output in H_i. The data in the multiplexer are selected with inputs S_3 and S_2. The other two data inputs to the multiplexer receive inputs A_{i-1} for the shift-right operation and A_{i+1} for the shift-left operation.

The above circuit must be repeated n times for an n-bit ALU.

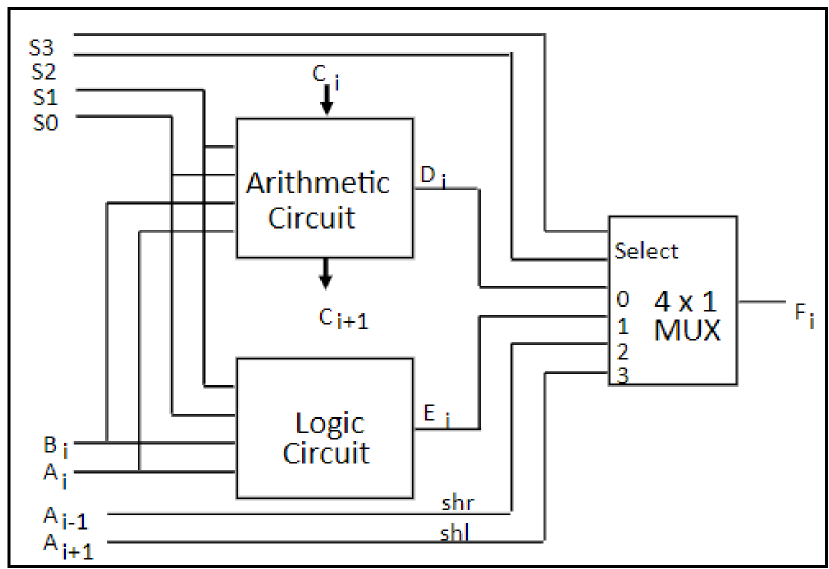


Figure 4.9 One stage of Arithmetic Logic Shift Unit

Table 4.6 lists the 14 operations of the ALU. The first eight are arithmetic operations and are selected with S_3S_2 = 00. The next four are logic operations and are selected with S_3S_2 = 01. The input carry has no effect during the logic operations and is marked with don't-care x 's. The last two operations are shift operations and are selected with S_3S_2 = 10 and 11. The other three selection inputs have no effect on the shift.

Table 4.6 Function Table for Arithmetic Logic Shift Unit

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer *A*
0	0	0	0	1	$F = A + 1$	Increment *A*
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement *A*
0	0	1	1	1	$F = A$	Transfer *A*
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement *A*
1	0	×	×	×	$F = \text{shr } A$	Shift right *A* into *F*
1	1	×	×	×	$F = \text{shl } A$	Shift left *A* into *F*

4.8 Summary

- The operations executed on data stored in registers are called ***microoperations***. A microoperation is an elementary operation performed on the information stored in one or more registers.
- The symbolic notation used to describe the microoperation transfers among registers is called a ***register transfer language (RTL)***.
- A more efficient scheme for transferring information between registers in a multiple-register configuration is a ***common bus system***. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time.

- A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a high-impedance state.
- The transfer of information from a memory word to the outside environment is called a ***read operation***. The transfer of new information to be stored into the memory is called a ***write operation***.

4.9 Key Terms

- **High-impedance state:** The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic signif-icance.
- **Binary Adder:** The digital circuit that generates the arithmetic sum of 2 binary numbers of any length is called Binary Adder.
- **Binary Adder- Subtractor:** The addition and subtraction operations can be combined into one common circuit by including an exclusive-OR gate with each full adder.
- **Logic Microoperations:** They are binary operations for strings of bits stored in registers.
- **Shift Microoperations:** They are used for the serial transfer of data.

4.10 Check Your Sum

Q1) What is Register Transfer Language? How does it support the Register Transfer Operation?

Q2) State the functioning of the Control function in Register transfer.

Q3) Explain the common bus system using multiplexers.

Q4) What is a Three-state Gate? State its application in designing a common bus system.

Q5) Write a short note on Logic Microoperations.

Q6) Discuss the types of shifts and their corresponding microoperations.

Q7) Explain Arithmetic Logic Shift Unit briefly.

References:

Computer System Architecture, M. Morris Mano.

Computer Organization and Architecture, *9th edition*, William Stallings, Pearson Publication.

MODULE: III

BASIC COMPUTER ORGANIZATION, DESIGN AND PROGRAMMING

Unit 5 – Basic Computer Organization and Design

Structure

5.0 Introduction

5.1 Unit Objectives

5.2 Instruction Codes

5.3 Register Sets

5.4 Instruction Sets

5.5 Machine Cycle, Timings, and Control

5.6 Input- Output Interface

5.7 Interrupts

5.8 Summary

5.9 Key Terms

5.10 Check Your Progress

5.0 Introduction

Any computer system is organized using some internal registers, timing and control unit, and a particular set of instructions to be executed by each processor of the system. As discussed earlier, a processor or microprocessor or central processing unit (CPU) is the main component of any computer. The need for advancement in the processor evolved with the technology and the need for high performance. The old generation processors were bound to be composed of different functional units for each operation, resulting in slow speed and big size. Although, the modern processors are mostly single-chip devices that are capable of supervising and executing all related tasks assigned to them. The data is processed by the processor and is stored in the memory devices. If there is a need to access the pre-stored data from the memory or the I/O devices, this is also accomplished by the processor itself. To transmit data from one place to another, buses are used. A certain set of registers is used in

the processor to perform all the functions including the arithmetic and logical operations.

5.1 Unit Objectives

This unit will help the reader to gain knowledge about:

- The basic architecture and design of the computer system.
- The concept of Instruction Codes, Register Sets, and Interrupts in the computer system.
- The machine cycle, timings, and control of the processor.

5.2 Instruction Codes

The processor can execute programs with multiple instructions. It should be noted that the instructions to be executed by the processor are always in the machine code. It is essential to convert the High-level language programs to machine language codes so that they are understandable by the computer. Generally, the machine codes are primitive and simple. For example, copy a data byte from external memory to the internal register or vice versa; add two numbers available within the processor registers. These instructions must be present in the binary form within the memory of the system so that they are accepted by the system.

To execute any type of instruction, the processor should perform the following steps:

- Fetch
- Decode
- Execute

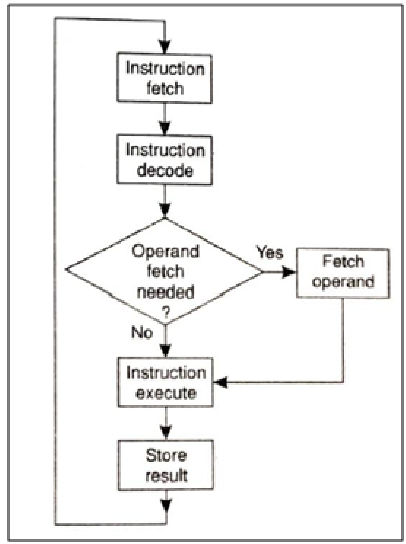


Figure 5.1 Flowchart for simplified instruction cycle

(**Source-** Computer Architecture and Organization, Subrata Ghoshal, Chapter-5, Page No. 99)

The combination of these three steps is called an *instruction cycle*. Flowchart for the simplified instruction cycle is shown in figure 5.1.

After fetching the instruction in the form of its Opcode and decoding it, the processor checks for any eventual operand fetch, which might be necessary for some instructions not for all instructions. Finally, the result of the instruction is stored and the whole cycle is repeated.

- **Instruction Fetch:** This is the first step for the instruction to fetch it from the external memory. This external memory is a vast area containing many bytes of instructions. Therefore, the processor must pinpoint the correct location of this large memory area to extract the target byte. Every memory location has a unique binary address. After receiving this address, the memory device decodes the address to locate the target byte and place it on the data bus, so that the content of that address is available to the processor.

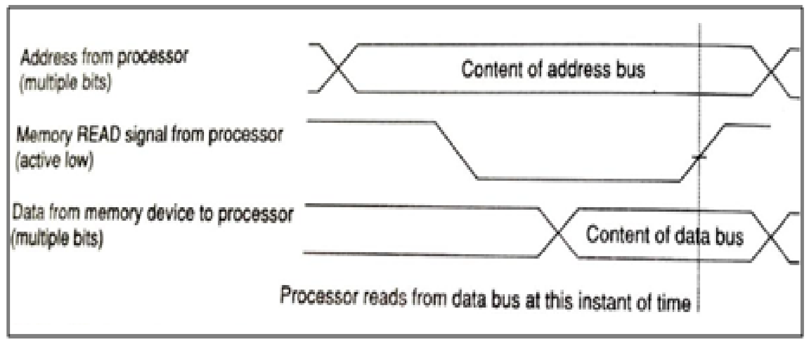


Figure 5.2 Timing diagram for instruction fetch

(**Source-** Computer Architecture and Organization, Subrata Ghoshal, Chapter-5, Page No. 100)

Therefore, for implementing the instruction fetch, the processor places an address, composed of multiple bits of binary information, on the address bus. At the same instance, a memory read signal is sent by the processor via control bus. When these signals come in contact with the memory device, the memory device automatically sends the data to the processor. On observing the above figure 5.2, the data must be stable when the memory read signal goes from low to high. At this stage, the address signals are stable to validate an effective data transaction.

- **Decode:** The next step after receiving the instruction code byte within itself is instruction decode, which is carried out within the processor itself. After completion of the instruction decoding, the processor knows whether to fetch operands from external memory or to increase a register by one or to store register content in an external memory location. This instruction decoding may be implemented through hardware. Instruction decoding may also be implemented through software which is known as micro-programming. This leads to a demand for a mini processor within the processor itself that should be completely dedicated for decoding & execution of the instruction
- **Execute:** After fetch and decoding instruction, this is the last and final phase of an instruction's execution. The instruction implementation for one or more instructions depends upon the instruction itself. After this

final step, the next instruction follows the process of fetch-decode-execute, and the operation continues.

The collection of several instructions, executed by the processor is known as the ***instruction set***. The instructions must contain information about the execution of any operation to be fed to the processor. The basic elements of a machine instruction are:

- **Operation Code/ Opcode:** An operation code stipulates the operation to be performed by the processor in a binary code format.
- **Source Operands:** The input variables of the operation on which the operation is performed are called source operands. There can be one or more source operands involved in the operation.
- **Result Operands:** The operands through which the result of the operation is generated are called result operands.
- **Next Instruction:** The next instruction element notifies the processor to fetch the next instruction after completing the execution of the present instruction.

5.3 Register Sets

Some internal registers are being offered by the processors for performing internal operations. These internal registers are a combination of several flip-flops and can store temporary information or some operands just like read/write memory. Most of these registers are user accessible while a few are not accessible to the user. However, the number of user-accessible registers varies from processor to processor. Those processors that are *memory oriented* offer a lesser number of internal registers as it expects the data or operands would mainly be stored and manipulated within the read/write memory (RAM) of the system. Although, some processors are said to be *register oriented* as they offer a larger number of registers. The program execution time for the processor would be less if the data are available within it rather than looking outside for them. However, complexity in instruction decoding increases with

more number of internal registers, as there is a demand for separate instruction for each register from the instruction set.

Different registers of the processor are:

- **Status Register:** Status registers display the results of arithmetic or logical operations performed by the processor. It depends on that particular operation that the result is zero or negative or generates a carry or odd/even parity. These types of status require additional care. Status register offers the status of the result of last arithmetic or logical operation with the help of pre-assigned bits. Each bit of the status register indicates one function of the result i.e. carry, parity, zero, overflow, and so on. They are known as flags that help the processor to take decisions about further action to be taken including branching of the program if required.
- **Accumulator:** The accumulator register acts as a location storage which holds an intermediate value on temporary basis. Earlier, only the accumulator used to store the result of all arithmetic or logical operations. But, now the result can be obtained initially in the accumulator and can be shifted to the desired memory location using proper instruction.
- **Program Counter:** This type of register is one of the most important registers within any processor, as it is responsible for holding the address of the memory location for the next instruction to be fetched by the processor. After fetching every instruction byte, this is automatically increased by one to point to the next byte. This auto-increment of the program counter is excluded in the case of program branching when a new value is reloaded in it. This counter is always initialized during system reset so that the first executable instruction byte is fetched from a predefined location of the memory.
- **Special Purpose Registers:** These are the internal registers of the processor that are not accessible to the user and are used at the time of

program execution. They include Memory Address Register (MAR), Memory Buffer Register (MBR), Instruction Register (IR).

The size and types of the register set differ from processor to processor according to their use and purpose.

5.3 Instruction Sets

In the machine language of the computer system, the instruction is represented as binary sequences. It is divided into fields corresponding to the elements of the instruction. For example, a 16-bit instruction can be divided into 3 fields with the first 4 bits for the Opcode, the next 6 bits for source operand, and the last 6 bits for the result operand, as shown in figure 5.3.

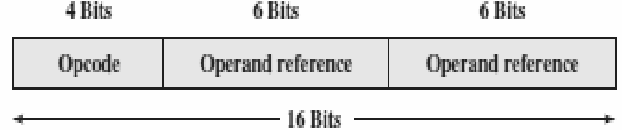


Figure 5.3 Simple Instruction format of 16 bits

(**Source-** Computer Organization and Architecture, Ninth Edition, William Stallings, Chapter-12, Page no. 408)

As the machine language codes are complex and difficult to understand, generally, symbolic representation of machine instructions is done. The Opcodes are denoted by mnemonics and the operands with certain symbols. For example, in the below instruction, ADD is the Opcode and P & X are the operands. The instruction means to add the content at location X to the contents of register P.

ADD P, X

In this instruction, X refers to the address of a location in the computer memory; P is a register containing some data. It is important to note that the operation is performed on the content of that location and not on the address of the location.

Instruction Types

Generally, the instructions are categorized in the following types:

- Data Move type instructions
- Data processing type instructions (Arithmetic and logical type instructions)
- Program flow and control type instructions

Data Move type instructions are often used when the data is copied from one location to another. The registers within the processor or external memory location or both in combination are considered to be the source and the destination of this data movement. It is interesting to note that the status of flags of the processor does not change due to these data move instructions. The data buses are chosen according to the movement of data, such as the external data bus is used when the data is transferred from the external memory location to the registers of the processor while the internal data bus can be used when there is internal communication in the processor. Different processors allow different data types among 8, 16, or 32 bits data.

Instruction	Function
Copy	Copy data from register to register
Load	Load data from memory to register. Load immediate data in register
Store	Save data from register to memory
Clear	Load register by 0s
Set	Load register by 1s
Exchange	Exchange data between two registers or between a register and a memory location
Push	Save data on stack-top indicated by stack pointer
Pop	Load data from stack-top into register
Input	Read data from input port and store within indicated register
Output	Write data from indicated register into output port

Figure 5.4 Data Move type instructions for the processor

(**Source-** Computer Architecture and Organization, Subrata Ghoshal, Chapter-6, Page No. 154)

The ***data processing instructions*** are meant to perform all the arithmetic and logical operations in the processor. Almost all the processors offer the four

basic arithmetic operations, Addition, subtraction, multiplication, and division of both signed and unsigned numbers; and the logical operations that are essential, AND, OR, NOT, XOR, shifting, and rotation operations. The status of the flag registers of the processor change with the execution of the instructions, depicting the further processing and the flow of the program.

The third major category of instructions is the ***program flow control type instructions***. It is assumed that the flow of data for execution is sequential i.e. the instructions loaded in consequent memory locations are executed one by one in a sequence, but this sequence can be changed by using other instructions, as per requirement. There are branch and jump operations used for controlling the program flow. The branch instruction is used to load a new value in the program counter so that the data stored at the new address is executed next instead of following the sequential order of the address location. The branch instruction is subdivided into conditional and unconditional branch instruction. The conditional branch instruction is executed only if the condition is satisfied while the unconditional branch instruction is irrespective of any condition. The conditional branching is decided by the status of the flags. We have already discussed the register set of the processor. Mainly, the carry and zero flags are used to decide the branching conditions. Subroutines are special functions pre-designed in the processor that are called whenever required through proper instructions.

Instruction	Function
Add	Add two operands
Add with carry	Add two operands and content of carry flag
Subtract	Subtract one operand from another
Subtract with carry	Subtract one operand from another along with carry
Multiply unsigned	Unsigned multiplication of two operands
Multiply signed	Signed multiplication of two operands
Divide unsigned	Unsigned division of two operands
Divide signed	Signed division of two operands
Increment by one	Increment an operand by one. Generally used for address manipulations.
Decrement by one	Decrement an operand bby one. Generally used for address manipulations.
Logical Operations	
AND	Logically AND two operands
OR	Logically OR two operands
XOR	Logically XOR two operands
NOT	Complement the operand
Shift right	Shift operand right one bit. LSB is lost. New constant introduced at MSB.
Shift left	Shift operand left one bit. MSB is lost. New constant introduced at LSB.
Rotate right	Shift operand right one bit. LSB is shifted to MSB.
Rotate left	Shift operand left one bit. MSB is shifted to LSB.
Compare	Compare two operands and reflect the result through flags
Test	Test flag bit(s)

Figure 5.4 Data Processing (arithmetic and logical) type instructions for the processor

(**Source-** Computer Architecture and Organization, Subrata Ghoshal, Chapter-6, Page No. 155)

Instruction	Function
Branch unconditional	Jump to the indicated address
Branch conditional	Test condition and if condition is true then jump to indicated address
Call a subroutine	Save PC on stack-top and branch to indicated address
Return from subroutine	Reload PC by address saved on stack-top
Return from interrupt	Enable interrupts and reload PC from stack-top
No operation	Do nothing
Wait	Wait for a signal input
Halt	Stop functioning of the processor
Skip next instruction	Execute the instruction immediately after the next instruction
Branch relative to PC	Add PC with an offset and branch there

Figure 5.5 Program flow and control type instructions for the processor

(**Source-** Computer Architecture and Organization, Subrata Ghoshal, Chapter-6, Page No. 155)

5.5 Machine Cycle, Timings, and Control

Machine Cycle

An instruction cycle comprises one or more machine cycles and every machine cycle consists of many T-states. A machine cycle is a step or time interval during which 1-byte of data is transacted between the processor and some external device. Generally, this external device is the memory device. However, in some exceptional cases, it might be an I/O device also. To transact the 1-byte of data, one machine cycle must be executed by the processor.

Each machine cycle is composed of many T-states. One complete oscillation of the processor clock is designated as one T-state. The number of T-states required to complete one machine cycle should be known by the processor. The relationship between machine cycle, T-states and instruction cycle is shown in figure 5.6

For example, the execution of an instruction increment a memory location by one is illustrated in figure 4.6. It is assumed that the first machine cycle fetches 1-byte instruction. As the data, to be increased by one, are available in the external memory location, the next machine cycle reads this operand from memory (means brings the data byte within the processor). The data are then increased by one by the processor and are stored back in the memory location in the third machine cycle.

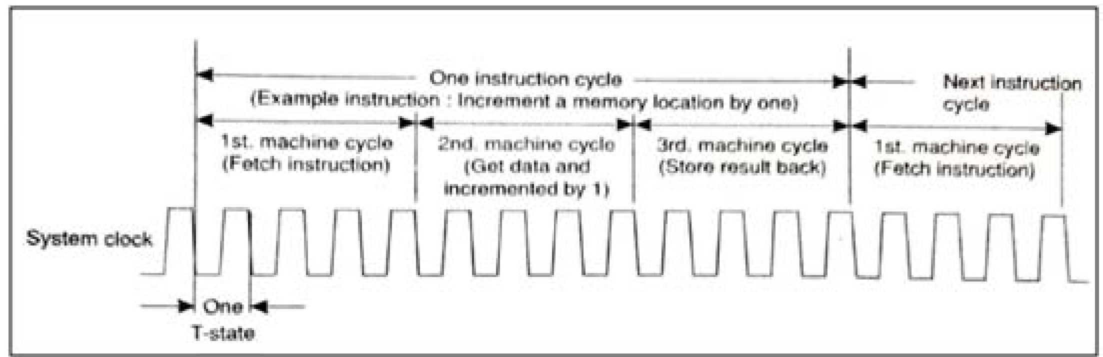


Figure 5.6 Relationship between instruction cycle, machine cycle, and T-state

(**Source-** Computer Architecture and Organization, Subrata Ghoshal, Chapter-5, Page No. 102)

Timings and Control

Timing and control play very important roles in the smooth and efficient functioning of any processor. For the further explanation of this concept, we will take the example of an ***interrupt.***

The interrupt may be introduced as an external asynchronous signal, which forces the processor to carry out something special for it by branching to a predefined address. An asynchronous program segment, known as ***interrupt service routine (ISR)*** is executed. It may be triggered at any time throughout the implementation of any instruction by the processor. However, an instruction's execution cannot be left half-way by the processor because it is already having such an interrupting signal.

Now, the solution to this type of problem would be, during the execution of each and every instruction, the processors reserve a specific time-slot for examining the existence of an interrupt signal. Let us take an example of the Intel 8085 processor. Next to the last T-state of the machine cycle of any instruction is reserved by the 8085 processor for checking this interrupt signal. If any interrupt is present, then there is no execution for the next instruction immediately and the interrupt ISR will be executed.

Essentially, timing and control are used to execute the instruction codes, register and computer instructions. In a basic computer, the timing for all the registers is controlled by a master clock generator. The signals for control are generated by a control unit. They provide control inputs for a multiplexer in a common bus (we know that a bus gets inputs from a multiplexer), control inputs in processor registers and micro-operations for an accumulator. The clock pulses from the master clock must be implemented to all the flip-flops and registers in the systems control unit. When the register is in the disabled mode, the clock pulse remains unchanged.

The control organization can be divided into two major types, the ***hardwired control*** and the ***micro-programmed control***. The main advantage of the hardwired control is that it can be optimized to obtain a result in a fast mode of operation. It is implemented with flip flops, gates, decoders, and other digital

circuits. If a change or modification is to be done in a hard wired control, it must be done to the wiring among various components.

On the other hand, the micro programmed control stores its control information in a control memory. To initiate the necessary set of micro operations, the control memory is programmed. The changes and modifications in a micro programmed control can be done by updating the micro-program in control memory.

5.6 Input- Output Interface

According to any processor, any device other than a memory device is an input/output (I/O) device. For example, devices like timer, interrupt controller, USART, DMA controller are found within the motherboard of any computer. However, from the user's point of view, I/O devices are something like a keyboard, printer, mouse, CRT, and so on. We will now discuss both types of peripheral units and study the method of communication necessary for the smooth operation of the computer.

All peripheral devices have their own processors within the devices. Therefore, communication between a host (computer) and any one of its peripheral units is essentially the communication between two processors. This communication link is always established through the wire-connections. This means that the processors are not placed within the same circuit and-may only be interconnected through some cable or multiple wires. The general structure of such an interface is shown in Figure 9.1. It can be observed that the processors in the motherboard of the host (computer) and all other processors within peripherals are generally different and have their own operating frequencies. Secondly, these processors are never directly interconnected but through some I/O ports. All necessary signals and data between any two processors are transferred through these ports. The method of data communication between the host and its peripheral may be any one of the following three:

- Programmed I/O

- Interrupt driven I/O
- Direct Memory Access (DMA)

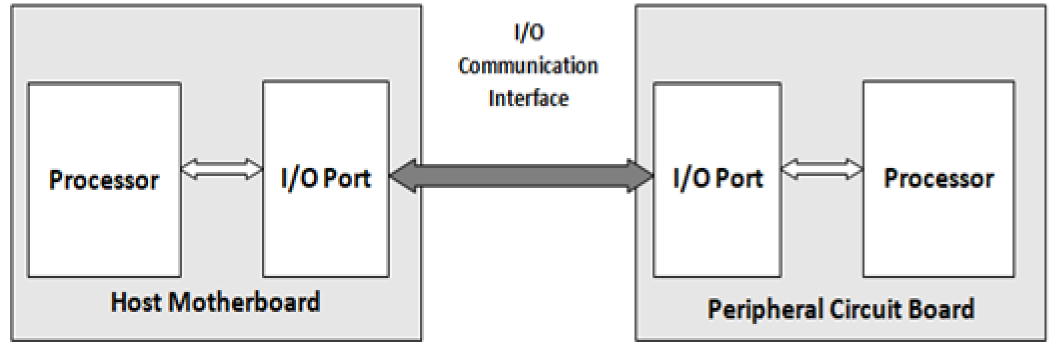


Figure 5.7 Representation of the link between a Host and its peripheral unit

(**Source-** Computer Architecture and Organization, Subrata Ghoshal)

The quantity of data transacted by the first two cases is comparatively lesser than that by the DMA. In the *programmed I/O*, the processor knows when to transmit, but generally has no idea when to receive the data from the external source and, therefore, sometimes the processor waits for data reception. In *interrupt-driven I/O*, the transaction begins with an interrupt from the peripheral device, which receives or transmits data from or to the host. Both of these types of transactions are controlled by the processor of the host. In the *DMA controller*, the processor of the host temporarily ends the system bus to the DMA controller, which takes care of the mass data transaction between a peripheral device and the memory of the host.

5.7 Interrupts

In all processors, some input pins are provided through which external devices can send signals to the processor to draw immediate attention. This attention may be necessary to receive a byte of data or to terminate a process or similar features related to the external device - which is sending the external signal. The major point here is, it must be recognized as an urgent request and the processor must leave everything whatsoever it was performing and must service the attention-drawing device immediately. These inputs of the processor

are known to be *interrupt inputs* and the process through which the immediate request is executed is known as *interrupt handling*. The minimum number of interrupt inputs is two while the maximum may be five or more, depending on processor to processor. It may be noted that after servicing an interrupt signal, the processor must resume its original work, which was left half-way because of the interrupt. This can be accomplished by utilizing system stack.

The modified instruction cycle including the interrupts is shown in figure 4.7.

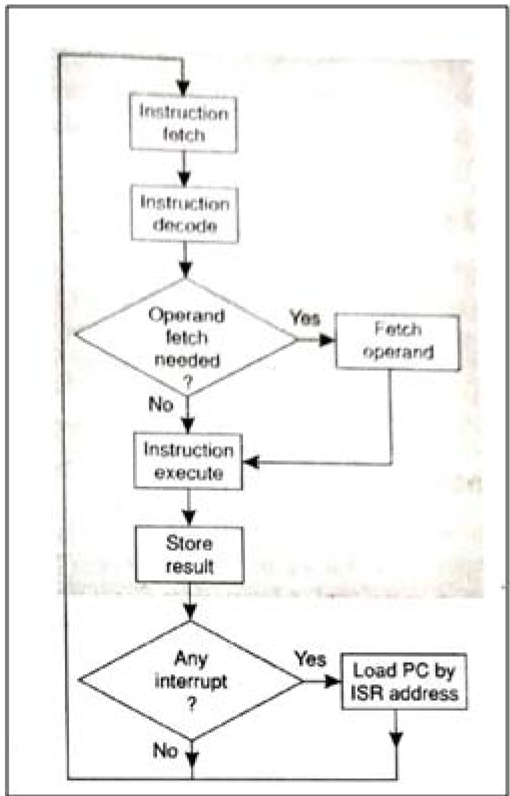


Figure 5.8 Modified flow- chart for simplified instruction cycle (with interrupt)

(**Source-** Computer Architecture and Organization, Subrata Ghoshal, Chapter-5, Page No. 103)

Interrupt Service Routine (ISR): The processor must execute a special routine developed to accommodate the need of the interrupting device to service an interrupt. These services are known as interrupt service routines (ISR) and terminated by a RETURN (or similar) instruction. Before branching to the interrupt service routine, as it is customary for the processor to store the

address of the next executable instruction on the stack-top, the execution of the RETURN instruction at the termination of ISR brings back the program control to its original point, which it had left at the time of receiving the interrupt. To conclude, an interrupt is an external hardware signal making a special request to the processor to execute a specific subroutine first, i.e., the ISR of the interrupting external device.

Vectored and Non-vectored Interrupts: The Interrupts are classified as vectored or non-vectored interrupts, on the basis of their pre-defined branching address. As the name suggests, vector quantities are associated with some directions. In case of vectored interrupts, the address of the interrupt is pre-defined, where ISR is to be located; while in case of non- vectored interrupts, there is no pre-defined branching address for the ISR of the interrupt. In such a case of non- vectored interrupt, the branching address is made available by the interrupting device itself.

Enabling, disabling, and Masking of Interrupts: Generally, the interrupts can be made enabled or disabled with the help of software commands. Moreover, in some processors, they may also be designated as masked or unmasked interrupts. After a system reset, all interrupts are disabled. They are enabled with the help of some software commands, whenever required. No external interrupt signal would be acknowledged if the interrupt is disabled.

Whenever any interrupt is acknowledged by the processor and branching for its ISR takes place, the processor automatically disables that interrupt to ensure that further successive branching for that interrupt is avoided. On avoiding this automatic disabling, the processor continues its branching to the same ISR repeatedly till the interrupting signal is active. For allowing further branching for the same interrupt, it becomes necessary to enable the interrupt, before executing the RETURN instruction. It should be assured by ISR that the original interrupting signal is no longer in existence, before enabling the interrupt.

One more advantage of masking includes preventing the processor to react to any interrupt. Non-maskable interrupts **(NMI)** are also considered by some

processors. The NMI are never disabled or masked and they react only to some extreme emergency conditions like power failure. There is a difference between disabling and masking an interrupt. During disabling an interrupt, the processor would never know about the existence of any eventual interrupting signal, while in case of making of interrupt, it can be traced. However, in either case, no automatic branching to the ISR would be allowed.

Types of Interrupts

There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

1. External interrupts
2. Internal interrupts
3. Software interrupts

External interrupts come from input-output (l/0) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source. Examples that cause external interrupts are l/0 device requesting transfer of data, l/0 device finished transfer of data, elapsed time of an event, or power failure. Timeout interrupt may result from a program that is in an endless loop and thus exceeded its time allocation. Power failure interrupt may have as its service routine a program that transfers the complete state of the CPU into a nondestructive memory in the few milliseconds before power ceases.

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps . Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation. These error conditions usually occur as a result of a premature termination of the instruc-tion execution. The service program that processes the internal interrupt deter-mines the corrective measure to be taken.

The difference between internal and external interrupts is that the inter-nal interrupt is initiated by some exceptional condition caused by the program itself rather than by an external event. Internal interrupts are synchronous with the program while external interrupts are asynchronous. If the program is

rerun, the internal interrupts will occur in the same place each time. External interrupts depend on external conditions that are independent of the program being executed at the time.

External and internal interrupts are initiated from signals that occur in the hardware of the CPU. A ***software interrupt*** is initiated by executing an instruc-tion. Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the super-visor mode. Certain operations in the computer may be assigned to the super-visor mode only, as for example, a complex input or output transfer procedure. A program written by a user must run in the user mode. When an input or output transfer is required, the supervisor mode is requested by means of a supervisor call instruction. This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode. The calling program must pass information to the operating system in order to specify the particular task requested.

5.8 Summary

- The processor is responsible to fetch, decode, and execute the instruction given to it.
- Programs developed with high-level language (HLL) are first changed to this machine code which is understandable by the processor.
- A machine cycle is a step or time interval during which 1-byte of data is transacted between the processor and some external device. One complete oscillation of the processor clock is designated as one T-state.
- The interrupt is an external asynchronous signal, which halts the operation of the processor and forces the processor to carry out a certain operation for it by branching to a predefined address and, thus,

executing a special program segment, known as *interrupt service routine (ISR)*. The interrupts may be enabled or disabled by software commands.

- Various types of registers of the processor are general-purpose registers, a memory address register, an instruction register, a memory buffer register, a status register, an accumulator, a program counter, and a stack pointer.

5.9 Key Terms

- **Accumulator:** It is a register in the processor that stores the result of arithmetic and logical operations performed in the ALU of the processor.
- **Memory Address Register:** It stores the address of the instruction that is to be fetched from the memory.
- **Memory Buffer Register:** This register stores the instruction received from or sent to the memory.
- **Interrupts:** The signals that are sent to the processor from the hardware or software to seek immediate attention and can halt the current process.

5.10 Check Your Progress

Q1) Explain the types of instructions in a computer system.

Q2) What is the basic function of an accumulator?

Q3) Discuss the basic operation of a processor with an example.

Q4) Explain the importance of the machine cycle in the functioning of any processor.

Q5) Write a short note on Interrupts.

Q6) Describe the concept of the Input- output interface in any processor.

Q7) Give details about the types of Registers and their functions.

References:

Computer System Architecture, M. Morris Mano

Computer Architecture and Organization, Subrata Ghoshal, Pearson Publication.

The essentials of Computer Organization and Architecture, Linda Null and Julia Lobur, Jones & Bartlett Learning.

Unit 6 – Programming the Basic Computer

Structure

6.0 Introduction
6.1 Unit Objectives
6.2 High Level, Assembly, and Machine Language
    6.2.1 High-Level Language
    6.2.2 Assembly Language
    6.2.3 Machine Language
6.3 Assembler
6.4 Programming Arithmetic and Logic Operations
6.5 Subroutines
6.6 Input- Output Programming
6.7 Summary
6.8 Key Terms
6.9 Check Your Progress

6.0 Introduction

Every processor of a computer system requires some software or some set of instructions for the execution of the data. Software and hardware both are equally important parts of a computer system. No computer system is feasible if any of them function inadequately. In earlier units, we have discussed the hardware characteristics of the computer system in detail, mainly related to the organizational or architectural part. The present unit focuses on the utility programs, arithmetic and logical operations, subroutines, and input-output programming of a computer system. The interaction between the hardware and software is governed by the instruction set of the processor. One can give the command to the computer with the help of a language which is comprehensible to the system. Suppose, there is a program written in a high- level language and is independent of the architecture of any processor. Before the execution of

such a program, it has to be translated into an assembly language program that is specific to a particular processor's architecture. Now, to proceed further, this assembly language program will again be converted into the digital machine codes that are accepted by the computer. These translations are carried out with the help of compilers and assemblers.

6.1 Unit Objectives

On completion of this unit, the reader will be able to gain knowledge about:

- Basics of high level, assembly, and machine language.
- Different types of assemblers.
- Programming of Arithmetic & Logic Operations.
- Concept of Subroutines.

6.2 High- Level, Assembly, and Machine Language

As already discussed, a computer system is based on certain languages, high-level, assembly, and machine language. Before discussing the various types of programming of arithmetic & logic operations, let's have a brief introduction about all the three types of languages of the computer system.

6.2.1 High- Level Language

The high-level languages are independent of any machine or processor. These languages use a simple format similar to english language so that it becomes easier for the user to develop computer programs in their own language. As they are system independent, so a high- level program can be translated into machine language and can run on any system. The high- level languages developed in the past time were COBOL and FORTRAN while the latest high-level languages used are C++, Java, etc. The same program can be written in two different high- level languages, but they have similar steps that are easily understood by the user. Each high-level language corresponds to its own set of keywords and syntax. The program before translation in the machine language is called ***source code*** and after translation, it is called ***machine or object***

code. The high- level language program is translated into machine code with the help of a ***Compiler***. The only drawback with using high- level languages is that they require translators for program execution and this process is time taking.

6.2.2 Assembly Language

An assembly language program is based on alphanumeric ***mnemonics*** *codes* other than using numeric machine codes of 0 and 1. It was developed to overcome the inconveniences of machine language. It is a machine-dependent or machine-specific language i.e. assembly language programs are developed according to the particular processor. The assembly language requires a translator to convert the program into machine language to be understood by the computer. This translator is known as ***Assembler.*** An assembly language program is easy to understand in comparison to the machine language program. Also, it is easier to locate, correct the errors, and modify the assembly language program. Some of the mnemonics codes used in assembly language programming are ADD for addition, START, SUB for subtraction, MOV for moving data, etc.

Assembly language basically lies between the high-level language programs and machine language programs. Therefore, in other words, it can be referred to as a low-level language program. Every assembly language is written for one particular processor. For instance, if A1 assembly language is written for the P1 processor then A1 assembly language cannot run on other processor P2.

The assembly programming language keeps a very strong concurrence between the high level (instructions) language and the machine language program, as it is dependent on the machine code instructions. In a high-level language, there are certain developed programs that are dependent on the machine. The reason for this is certainly because of the translation of a high level to its complementary assembly language. This translation or in computer architecture terms, the translation is called a compilation, which is further termed as the ***compiler.*** Not only in the high-level language programs but also

in assembly language, the programs developed are also translated to the machine language. Here the 'translated' refers to assembled which is done by an ***assembler.***

6.2.3 Machine Language

Machine language was the foremost programming language developed in the past. A program written in the series of 0's and 1's is known as a machine language program. It is the only language understood by the computer without using any translator. It consists of two parts: ***operation code*** and ***operand***. The operation code commands the computer for what operation is to be performed and an operand shows the path where the computer can store the data to be executed. Thus, a machine language program is a combination of numeric codes for the instruction and location of the storage for data. It is obvious that it is machine-specific i.e. it is different for different processors. It is complex and difficult for users to write programs in machine language, as a result, more errors are there and it is difficult to modify the programs.

6.3 Assembler

The assembler is a software utility that takes input as an assembly program and produces object code as output. In simple words, programs in assembly language programs are developed by translating its instruction set into the machine language program using any processor (the processor of assembly language and machine language must be the same). This process is done by the assembler.

In any assembler, the input is denoted by the *source code* and output is denoted by the *object code.* To write and save the assembly language program, generally, an *editor* is used by most of the assemblers. In the absence of an editor, any word processor can also be used to write the assembly language program. The output of the assembler consists of an executable version of the input (source code) in the machine language program. This source code is scrutinized by the assembler line by line (one at a time) including all labels,

constants, and variables, which are reserved and assigned appropriate addresses to hold their values. A certain table is created for this purpose which is known as a **symbol table** or *SymTab*. After the scan is complete, a code is generated by the assembler which is the output or the object code in binary form and it may be executed directly by the assembler.

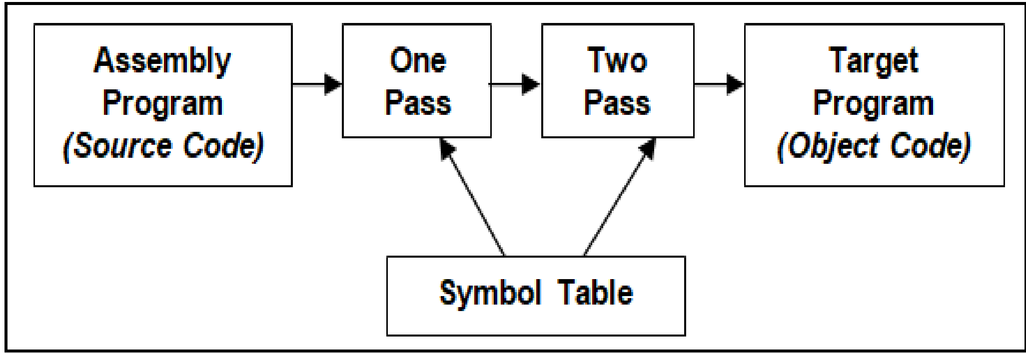


Figure 6.1 Basic working of an Assembler

There are two types of assemblers: *two-pass assembler* and *one-pass assembler.*

- **Two-pass Assembler**

  A two- pass assembler takes two cycles of scanning of the whole source code. For the first pass, a symbol table is generated that contains user-defined symbols representing their binary equivalent value. Binary translation occurs during the second pass. *Location counter (LC)* is a memory word used to track the location of the instructions. LC stores the address of memory location of the instruction or the operand currently in process. LC is initialized using the ORG pseudo instruction with the value of first location or to 0. As the locations of the instructions are in a sequence so LC is incremented by one after processing each line. Figure 6.2 represents the flow chart describing the first pass of an assembler.

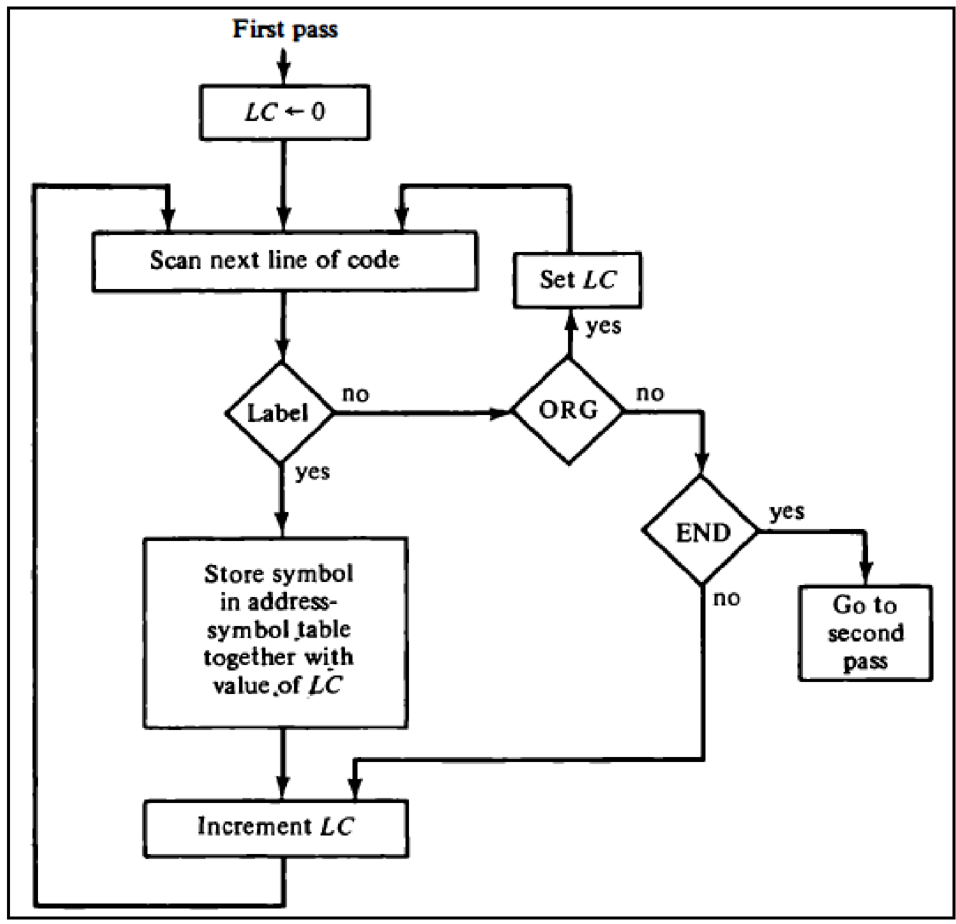


Figure 6.2 Flow chart denoting first pass of an assembler

(Source- Computer System Architecture, Morris Mano third edition, Chapter-6, Page no. 186)

- **One-pass Assembler**

  When there are two separate pathways used in generating the object code from the source code, some of the assemblers use only one pathway, unlike, two-pass assembler. Here, instructions of all the machine codes are generated in pass-one only. During the one-pass the assembler proceeds with both label definitions and assembly.

6.4 Programming Arithmetic and Logic Operations

The computer system works on a set of certain instructions to perform certain operations. Some systems use only one machine instruction while others may

perform the operation using more than one machine instruction. Considering the four basic arithmetic operations, machine instructions are available for either all four of them or basic computers have instructions for only one operation i.e. addition. Certain programs are designed for designated operations. When a set of instructions including a program is used for an operation then it is implemented using software and when only one instruction is used for one operation then hardware is used to implement such operation. Due to complex additional circuitry and high cost, hardware implementation is less preferred and dependency is more on software implementation. We here focus on the software implementation and designing of programs for arithmetic and logical operations.

Let us consider multiplication of two 8- bit numbers and develop a program for the same. For simplification, the sign bit of both the numbers is neglected. The procedure of the multiplication is the same as discussed before. The bits of the multiplier Y are checked and added to the multiplicand X as many times as there are 1's in Y and shifting the value of X to the left. P stores the intermediate sums also known as ***partial product*** that start with zero. The multiplicand X is added to the partial product P for each bit of Y is 1 and the X is shifted to left after checking the bits of multiplier Y. The final product can be obtained from the final value of P.

To generate a program for the above operation, a counter CTR is set to -8 as we are considering both multiplicand and multiplier to be of 8 bits so the program will traverse 8 times. The value of Y is loaded in the accumulator AC, circulating the register E and AC to the right and the value of shifted number is stored back into Y. E contains the lower- order bit of the multiplier, if the value of E is 1, the value of multiplicand X is added to the partial product denoted as P; if the value of E is 0, there is no change in the partial product. Then, the value of X is shifted to the left after loading it into AC and circulating E and AC to the left. This particular loop is being repeated for 8 times and the CTR counter is incremented by 1 each time. When the CTR value reaches zero, the loop is stopped and the final product is stored in P.

The program in Table 6.1 lists the instructions for multiplying two unsigned numbers. The initialization is not listed but should be included when the program is loaded into the computer. The initialization consists of bringing the multiplicand and multiplier into locations X and Y, respectively; initializing the counter to - 8; and initializing location P to zero. If these locations are not initialized, the program may run with incorrect data. The program itself is straightforward and follows the steps listed in the flowchart. The comments may help in following the step-by-step procedure.

Table 6.1 Program to Multiply Two Positive Numbers

```
        ORG 100
LOP,    CLE         /Clear E
        LDA Y       /Load multiplier
        CIR         /Transfer multiplier bit to E
        STA Y       /Store shifted multiplier
        SZE         /Check if bit is zero
        BUN ONE     /Bit is one; go to ONE
        BUN ZRO     /Bit is zero; go to ZRO
ONE,    LDA X       /Load multiplicand
        ADD P       /Add to partial product
        STA P       /Store partial product
        CLE         /Clear E
ZRO,    LDA X       /Load multiplicand
        CIL         /Shift left
        STA X       /Store shifted multiplicand
        ISZ CTR     /Increment counter
        BUN LOP     /Counter not zero; repeat loop
        HLT         /Counter is zero; halt
CTR,    DEC -8      /This location serves as a counter
X,      HEX 000F    /Multiplicand stored here
Y,      HEX 000B    /Multiplier stored here
P,      HEX 0       /Product formed here
        END
```

This example has shown that if a computer does not have a machine instruction for a required operation, the operation can be programmed by a sequence of machine instructions. Thus we have demonstrated the software implementation of the multiplication operation.

Double-Precision Addition

When two 16-bit unsigned numbers are multiplied, the result is a 32-bit product that must be stored in two memory words. A number stored in two memory words is said to have double precision. When a partial product is computed, it is necessary that a double-precision number be added to the shifted multiplicand, which is also a double-precision number. For greater accuracy, the programmer may wish to employ double-precision numbers and perform arithmetic with operands that occupy two memory words.

In a double- precision addition, one of the double-precision numbers is placed in two consecutive memory locations, AL and AH, with AL holding the 16 low-order bits. The other number is placed in BL and BH. The two low-order portions are added and the carry transferred into E. The AC is cleared and the bit in E is circulated into the least significant position of the AC. The two high-order portions are then added to the carry and the double-precision sum is stored in CL and CH.

Logical Operations

The basic computer has three machine instructions that perform logic operations: ***AND, CMA, and CLA***. The LDA instruction may be considered as a logic operation that transfers a logic operand into the AC. There are 16 logic operations that can be implemented by software means because any logic function can be implemented using the AND and complement operations. For example, the OR operation is not available as a machine instruction in the basic computer. From DeMorgan's theorem we recognize the relation x + y = (x 'y')'. The second expression contains only AND and complement operations. A program that forms the OR operation of two logic operands A and B is as follows:

LDA A	Load First Operand A
CMA	Complement to get $\underline{A}$
STA TMP	Store in a temporary Location

LDA B	Load Second Operand B
CMA	Complement to get $\overline{B}$
AND TMP	AND with $\overline{A}$ to get $\overline{A} \wedge \overline{B}$
CMA	Complement again to get A ∨ B

The other logic operations can be implemented by software in a similar fashion.

Shift Operations

The circular-shift operations are machine instructions in the basic computer. The other shifts of interest are the logical shifts and arithmetic shifts. These two shifts can be programmed with a small number of instructions. The logical shift requires that zeros be added to the extreme positions. This is easily accomplished by clearing E and circulating the AC and E.

For a logical shift-right operation we need the two instructions: ***CLE*** and ***CIR***

For a logical shift-left operation we need the two instructions: ***CLE*** and ***CIL***

The arithmetic shifts depend o n the type of representation of negative numbers. For the basic computer we have adopted the signed-2's complement representation. For an arithmetic right-shift it is necessary that the sign bit in the leftmost position remain unchanged. But the sign bit itself is shifted into the high-order bit position of the number.

The program for the arithmetic right-shili requires that we set ***E*** to the same value as the sign bit and circulate right, thus:

```
CLE   /Clear E to 0
SPA   /Skip if AC is positive; E remains 0
CME   /AC is negative; set E to 1
CIR   /Circulate E and AC
```

For arithmetic shift-left it is necessary that the added bit in the least significant position be 0. This is easily done by clearing E prior to the circulate-left operation. The sign bit must not change during this shift. With a circulate

instruction, the sign bit moves into E. It is then necessary to compare the sign bit with the value of E after the operation. If the two values are equal, the arithmetic shift has been correctly implemented. If they are not equal, an overflow occurs. An overflow indicates that the unshifted number was too large. When multiplied by 2 (by means of the shift), the number obtained exceeds the capacity of the AC .

6.5 Subroutines

Certain tasks cannot be performed by the program alone. They need additional routines known as ***subroutines***. Subroutines are routines that can be called within the main body of the program at any point of time. There are cases when many programs contain identical code sections. These code sections can be saved in subroutines and used wherever common codes are used.

All the microprograms that implement subroutine must have extra memory space to store the return address. The extra space is called the ***subroutine register***. It is used to store the return address during a subroutine call and restore during subroutine return. The incremented output must be placed in a subroutine register from a CAR. This must be branched to the beginning of subroutine. Now, the register becomes a means of transferring addresses to return to the main routine. The registers must be arranged in the LIFO (Last In First Out) stack so that it is easy to get the addresses.

Programs are often required to perform the same operation many times. For example, applications involving integers are often required to display integer values. The code fragment that performs an operation must be executed each time the operation is to be performed. One way to achieve this would be to duplicate the code fragment at each point it is needed. This could work, but might make the program very large. A better solution might be to load one copy of the code fragment into memory as a subroutine, and to provide instructions that allow a program to invoke (i.e. transfer control to) the subroutine whenever the operation is to be performed.

Instead of repeating the code every time it is needed, there is an obvious advantage if the common instructions are written only once. Each time that a subroutine is used in the main part of the program, a branch is executed to the beginning of the subroutine. After the subroutine has been executed, a branch is made back to the main program.

A subroutine consists of a self-contained sequence of instructions that carries out a given task. A branch can be made to the subroutine from any part of the main program. This poses the problem of how the subroutine knows which location to return to, since many different locations in the main program may make branches to the same subroutine. It is therefore necessary to store the return address somewhere in the computer for the subroutine to know where to return. Because branching to a subroutine and returning to the main program is such a common operation, all computers provide special instruc-tions to facilitate subroutine entry and return.

In the basic computer, the link between the main program and a sub-routine is the ***BSA instruction*** (branch and save return address). To explain how this instruction is used, let us write a subroutine that shifts the content of the accumulator four times to the left. Shifting a word four times is a useful operation for processing binary-coded decimal numbers or alphanumeric char-acters. Such an operation could have been included as a machine instruction in the computer. Since it is not included, a subroutine is formed to accomplish this task. The program represented in Table 6.2 starts by loading the value of X into the AC. The next instruction encountered is BSA SH4. The BSA instruction is in location 101 . Subroutine SH4 must return to location 102 after it finishes its task. When the BSA instruction is executed, the control unit stores the return address 102 into the location defined by the symbolic address SH4 (which is 109). It also transfers the value of SH4 + 1 into the program counter. After this instruction is executed, memory location 109 contains the binary equivalent of hexadecimal 102 and the program counter contains the binary equivalent of hexadecimal lOA. This action has saved the

return address and the subroutine is now executed starting from location lOA (since this is the content of PC in the next fetch cycle).

Table 6.2 Program to Demonstrate the Use of Subroutines

Location			
		ORG 100	/Main program
100		LDA X	/Load X
101		BSA SH4	/Branch to subroutine
102		STA X	/Store shifted number
103		LDA Y	/Load Y
104		BSA SH4	/Branch to subroutine again
105		STA Y	/Store shifted number
106		HLT	
107	X,	HEX 1234	
108	Y,	HEX 4321	
			/Subroutine to shift left 4 times
109	SH4,	HEX 0	/Store return address here
10A		CIL	/Circulate left once
10B		CIL	
10C		CIL	
10D		CIL	/Circulate left fourth time
10E		AND MSK	/Set *AC*(13–16) to zero
10F		BUN SH4 I	/Return to main program
110	MSK,	HEX FFF0	/Mask operand
		END	

The computation in the subroutine circulates the content of AC four times to the left. In order to accomplish a logical shift operation, the four low-order bits must be set to zero. This is done by masking FfFO with the content of AC. A mask operation is a logic AND operation that clears the bits of the AC where the mask operand is zero and leaves the bits of the AC unchanged where the mask operand bits are 1's.

The last instruction in the subroutine returns the computer to the main program. This is accomplished by the indirect branch instruction with an address symbol identical to the symbol used for the subroutine name. The address to which the computer branches is not SH4 but the value found in location SH4 because this is an indirect address instruction. What is found in

location SH4 is the return address 102 which was previously stored there by the BSA instruction. The computer returns to execute the instruction in loca-tion 102. The main program continues by storing the shifted number into location X. A new number is then loaded into the AC from location Y, and another branch is made to the subroutine. This time location SH4 will contain the return address 105 since this is now the location of the next instruction after BSA. The new operand is shifted and the subroutine returns to the main program at location 105.

From the above example, it should be noted that the first memory location of each sub-routine serves as a link between the main program and the subroutine. The procedure for branching to a subroutine and returning to the main program is referred to as a subroutine ***linkage***. The BSA instruction performs an opera-tion commonly called subroutine ***call***. The last instruction of the subroutine performs an operation commonly called subroutine ***return***. The procedure used in the basic computer for subroutine linkage is commonly found in computers with only one processor register. Many com-puters have multiple processor registers and some of them are assigned the name ***index registers***. In such computers, an index register is usually employed to implement the subroutine linkage. A branch-to-subroutine instruction stores the return address in an index register. A return-from-subroutine in-struction is affected by branching to the address presently stored in the index register.

6.6 Input- Output programming

Users of the computer write programs with symbols that are defined by the programming language employed. The symbols are strings of characters and each character is assigned an 8-bit code so that it can be stored in computer memory. A binary-coded character enters the computer when an INP (input) instruction is executed. A binary-coded character is transferred to the output device when an OUT (output) instruction is executed. The output device detects the binary code and types the corresponding character.

Table 6.3(a) lists the instructions needed to input one character and store it in memory. The SKI instruction checks the input flag to see if a character is available for transfer. The next instruction is skipped if the input flag bit is 1. The INP instruction transfers the binary-coded character into AC(0-7). The character is then printed by means of the OUT instruction. A terminal unit that communicates directly with a computer does not print the character when a key is depressed. To type it, it is necessary to send an OUT instruction for the printer. In this way, the user is ensured that the correct transfer has occurred. If the SKI instruction finds the flag bit at 0, the next instruction in sequence is executed. This instruction is a branch to return and check the flag bit again. Because the input device is much slower than the computer, the two instruc-tions in the loop will be executed many times before a character is transferred into the accumulator.

Table 6.3(b) lists the instructions needed to print a character initially stored in memory. The character is first loaded into the AC. The output flag is then checked. If it is 0, the computer remains in a two-instruction loop checking the flag bit. When the flag changes to 1, the character is transferred from the accumulator to the printer.

Table 6.3 Programs to Input and Output One Character

```
(a) Input a character:
   CIF,     SKI           /Check input flag
            BUN CIF       /Flag=0, branch to check again
            INP           /Flag=1, input character
            OUT           /Print character
            STA CHR       /Store character
            HLT
   CHR,     —             /Store character here
(b) Output one character:
            LDA CHR       /Load character into AC
   COF,     SKO           /Check output flag
            BUN COF       /Flag=0, branch to check again
            OUT           /Flag=1, output character
            HLT
   CHR,     HEX 0057      /Character is "W"
```

A computer is not just a calculator but also a symbol manipulator. The binary coded characters that represent symbols can be manipulated by computer instructions to achieve various data-processing tasks. One such task may be to pack two characters in one word. This is convenient because each character occupies 8 bits and a memory word contains 16 bits.

The running time of input and output programs is made up primarily of the time spent by the computer in waiting for the external device to set its flag. The waiting loop that checks the flag keeps the computer occupied with a task that wastes a large amount of time. This waiting time can be eliminated if the interrupt facility is used to notify the computer when a flag is set. The advan-tage of using the interrupt is that the information transfer is initiated upon request from the external device. In the meantime, the computer can be busy performing other useful tasks. Obviously, if no other program resides in memory, there is nothing for the computer to do, so it might as well check for the flags. The interrupt facility is useful in a multiprogram environment when two or more programs reside in memory at the same time. Only one program can be executed at any given time even though two or more programs may reside in memory. The interrupt facility allows the running program to proceed until the input or output device sets its ready flag.

6.7 Summary

- Every processor of a computer system requires some software or some set of instructions for the execution of the data.
- The program before translation in the machine language is called ***source code*** and after translation, it is called ***machine or object code***.
- The high- level language program is translated into machine code with the help of a ***Compiler***.
- The assembly language requires a translator to convert the program into machine language to be understood by the computer. This translator is known as ***Assembler.***

- Subroutines are routines that can be called within the main body of the program at any point of time.

6.8 Key Terms

- **Compiler:** It is a computer software that translates (compiles) source code written in a high-level language (e.g., C++) into a set of machine-language instructions.
- **Assembler:** It is a computer software that translates an assembly language program into a set of machine-language instructions.
- **Symbol Table:** It is a data structure used by a language translator such as a compiler or interpreter, where each identifier (or symbol) in a program's source code is associated with information relating to its declaration or appearance in the source.
- **BSA instruction:** In the basic computer, the link between the main program and a sub-routine is the BSA instruction (branch and save return address).

6.9 Check Your Progress

Q1) Explain the concept of subroutines with an example.

Q2) Define Assembler. Also explain the types of assemblers.

Q3) Write a short note on Input-Output Programming.

Q4) Write a program to demonstrate the use of subroutines.

Q5) Discuss double precision addition with an example.

References:

Computer System Architecture, M. Morris Mano

Computer Architecture and Organization, Subrata Ghoshal, Pearson Publication.

MODULE: IV

CENTRAL PROCESSING UNIT AND MEMORY ORGANIZATION

Unit 7 – Central Processing Unit

Structure

7.0 Introduction
7.1 Unit Objectives
7.2 General Register Organization
7.3 Stack Organization
7.4 Instruction Formats
7.5 Addressing Modes
7.6 Types of Instructions
7.7 Reduced Instruction Set Computer (RISC)
7.8 Summary
7.9 Key Terms
7.10 Check Your Progress

7.0 Introduction

The part of the computer that performs the bulk of data-processing operations is called the Central Processing Unit (CPU). CPU is said to be the brain of a computer system. The CPU along with the memory and the I/O subsystems develops a powerful computer system. The CPU is made up of three major parts, as shown in figure 7.1 The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required microoperations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

The CPU performs a variety of functions dictated by the type of instructions that are incorporated in the computer. Computer architecture is sometimes defined as the computer structure and behavior as seen by the programmer that uses machine language instructions. This includes the instruction

formats, addressing modes, the instruction set, and the general organization of the CPU registers.

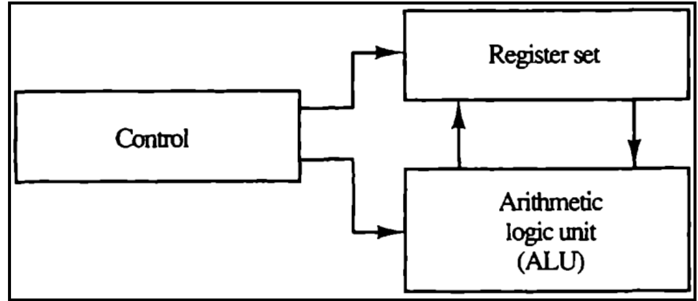


Figure 7.1 Major Components of CPU

Companies such as AMD, IBM, Intel, Motorola, SGI, and Sun manufacture CPUs that are used in various kinds of computers such as desktops, mainframes, and supercomputers. A CPU comprises thin layers of thousands of transistors. Transistors are microscopic bits of material that block electricity at one voltage (non-conductor) and allow electricity to pass through them at different voltage (conductor). These tiny bits of materials are the semiconductors that take two electrical inputs and generate a different output when one or both inputs are switched on. As CPUs are small, they are also referred to as microprocessors. Modern CPUs are called integrated chips. It is so called because several types of components such as execution core, Arithmetic Logic Unit (ALU), registers, instruction memory, cache memory, and the input/output controller are integrated into a single piece of silicon. This unit illustrates the basic functioning of the CPU.

7.1 Unit Objectives

On completion of this unit, the reader will be able to gain knowledge about:

- Basic functions of CPU.
- General Register Organization in CPU.
- Stack Organization.
- Instruction Format in CPU.

7.2 General Register Organization

As already discussed, memory locations are needed for storing pointers, counters, return addresses, tempo-rary results, and partial products during multiplication. Having to refer to memory locations for such applications is time consuming because memory access is the most time-consuming operation in a computer. It is more convenient and more efficient to store these intermediate values in processor regis-ters. When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while perform-ing various microoperations. Hence it is necessary to provide a common unit that can perform all the arithmetic, logic, and shift microoperations in the processor.

A group of flip-flops form a register. A register is a special high speed storage area in the CPU. They comprise combinational circuits that perform data processing. The data is always represented in a register before processing. The registers speed up the execution of programs.

Registers perform two important functions in the CPU operation. They are:

1. Providing a temporary storage area for data. This helps the currently executing programs to have a quick access to the data, if needed.
2. Storing the status of the CPU as well as information about the currently executing program.

A bus organization for seven CPU registers is shown in figure 7.2. The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses from the inputs to a common arithmetic logic unit (ALU). The operation selected in the ALU deter-mines the arithmetic or logic microoperation that is to be performed. The result of the microoperation is available for output data and also goes into the inputs of all the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs,

thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

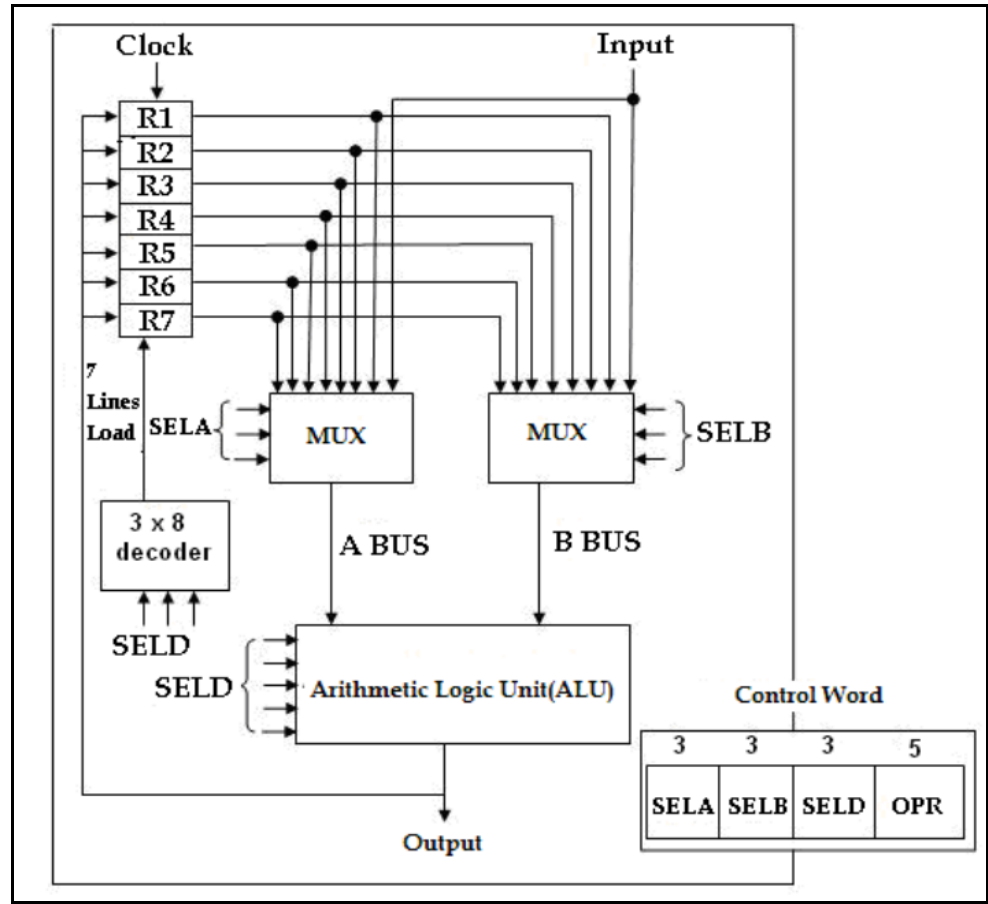


Figure 7.2 General Organization of Registers

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation R 1←R2 + R3 the control must provide binary selection variables to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.
2. MUX B selector (SELB): to place the content of R3 into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition A + B.
4. Decoder destination selector (SELD): to transfer the content of the output bus into R1.

The four control selection variables are generated in the control unit and must be available at the beginning of a clock cycle. The data from the two source registers propagate through the gates in the multiplexers and the ALU, to the output bus, and into the inputs of the destination register, all during the clock cycle interval. Then, when the next clock transition occurs, the binary informa-tion from the output bus is transferred into R1. To achieve a fast response time, the ALU is constructed with high-speed circuits.

Control Word: There are 14 binary selection inputs in the unit, and their combined value specifies a control word. The 14-bit control word is defined in figure 7.2. It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a register for the B input of the ALU. The three bits of SELD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular microoperation.

Table 7.1 Encoding of Register Selection Fields

Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

The encoding of the register selections is specified in Table 7.1. The 3-bit binary code listed in the first column of the table specifies the binary code for each of the three fields. The register selected by fields SELA, SELB, and SELD is the one whose decimal number is equivalent to the binary number in the code. When SELA or SELB is 000, the corresponding multiplexer selects the external input data. When SELD = 000, no destination register is selected but the contents of the output bus are available in the external output.

7.3 Stack Organization

The stack is an area within the system RAM reserved for some special storage by the program or programmer. In other words, the stack consists of several bytes of the read/write memory where some special data may be stored in and restored form, as per the programmer's requirements. This cannot be accomplished by using the available registers within the processor because the number of registers is very limited and they have their other specific purpose rather than storing return addresses.

The stack is generally used to store some important addresses and data sets. The particular location within the stack is known as the ***stack-top.*** Usually, the address of this stack top is available in the register designated for this specific purpose and hence known as the ***stack pointer.***

Stack as Storage Area: Generally, two types of instructions are offered by every processor to handle the stack directly. These two instructions are PUSH and POP. PUSH instructions place the data on the stack-top and POP instruction takes it out from the stack-top. This data may be available in a general-purpose register. The registers within the processor, which holds the current stack-top address, is assigned as a stack pointer (SP). The stack pointer (SP) is automatically changed by the processor, whenever any data is placed on the top of the stack or taken out from it, PUSH operation increments the SP while SP is decremented for a POP instruction.

Subroutines and Stack: Subroutines are the pre-defined functions that can be called whenever required. There is an ample use of stacks for subroutine calls.

In such cases, the return address of the subroutine is kept on the stack-top before branching to this subroutine. The subroutines are terminated by a RETURN instruction. On completion of the execution of the subroutine, this RETURN instruction commands the processor to load the program counter from the stack-top, thus returning to the original part of the program that was left to branch to the subroutine.

7.4 Instruction Format

We already know that the execution of the instruction results in a machine language code (binary form). This machine code is known as the ***Opcode*** of the processor that occupies one byte or sometimes multiple bytes of memory for the description of any instruction. The format for representing such Opcodes varies with the type of instruction. Such as the instructions like RETURN or RET do not need any operand and occupies one byte of memory while the instructions of immediate addressing mode may require two bytes of memory (one byte for Opcode or details of the destination register and one byte for the data to be loaded in the register). Following features differentiate the instruction sets from each other:

- The storage of operands (stack structure or registers or both are used to store the data).
- The number of operands per instruction.
- Location of the operand
- Type and size of the operand (can be numbers, addresses, or characters)

To finalize the instruction format, the architecture of the concerned processor plays an important role. For an 8-bit processor (with an 8-bit data bus), instructions would be necessary to load 8-bit data. In the majority of such cases, provision to handle 16-bit data also needs to be incorporated. In some cases, bit or nibble (4 bits) handling provisions also may be ruled out.

Furthermore, the number of available registers within the processor would intervene in the instruction formatting. If the number of on-chip registers is more, a larger number of data bits must be assigned to target one of these

registers by decoding the encoded register-addressing bits. In other words, the width of the register field within the instruction code (Opcode) would increase.

The instruction set plays an important role to finalize the instruction format. If the number of instructions is more (CISC processor) then more bits are necessary to encode the instruction. On the other hand, lesser number of bits are required in encoding the instructions due to less number of instruction types in RISC processors.

Instruction format and field are the two different aspects of any instruction. The field refers to the number of bits necessary to specify a particular parameter like an operand's address. The format of an instruction indicates the number and order of these fields, including the Opcode. In general, an opcode is placed at the beginning of the instruction.

7.5 Addressing Modes

Addressing modes display the method by which the data is targeted by the instruction. These modes are generally related to the data transfer instructions. It specifies a protocol for modifying the address of the instruction before the operand is actually referred. The addressing modes provide the following functions like pointers to memory, indexing of data, counters for loop control, etc. to make the programming more adaptable. They also reduce the number of bits in the addressing field of instruction. Addressing modes are important for the architecture level of the instruction set and for instruction decoding. The instruction decoding process becomes complex if the number of addressing modes is increased for instruction. This is the reason why RISC processors have less number of addressing modes to reduce the complexity in decoding the instructions of the processor. An effective address is considered to be the memory address or the address of the operand obtained from the execution of computational instruction indicated by given addressing mode. Let us discuss some of the widely used addressing modes by the different processors.

1. **Implied Addressing Mode:** In the implied addressing mode, the operands are indicated implicitly in the meaning of the instruction only.

Generally, the instructions that include the accumulator follow the implied mode of addressing the instructions. For example, the instruction 'complement accumulator' (CMA) follows an implied addressing mode as the operand in the accumulator is implied in the definition of the instruction itself.

2. **Immediate Addressing Mode:** In this mode, the operand is included in the instruction itself, i.e. the instruction has an operand field instead of having an address field. This addressing mode is generally used for loading '*constants*' in the processor. For example, MVI B, 30H. This implies that the constant value 30H is to be moved in the register B using this instruction. Since the targeted operand is included in the instruction itself, such instructions follow immediate addressing mode.
3. **Direct Addressing Mode:** In this addressing mode, the present address of data is included in the instruction. The operand is in the memory and the address of the operand is provided directly by the address field in the instruction. For example, LDA 200H. This instruction will load the accumulator directly with the data of the memory location of address 200H.

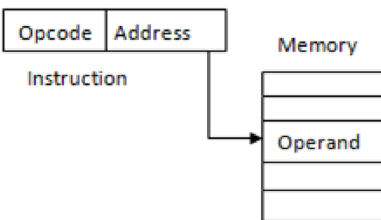


Figure 7.3 Direct Addressing Mode

4. **Register Direct Addressing Mode:** This mode corresponds to the operands residing in the CPU registers. The particular register containing the operand is selected from the register field of the instruction. For example, MOV A, B instruction will move the data of register B to Accumulator.

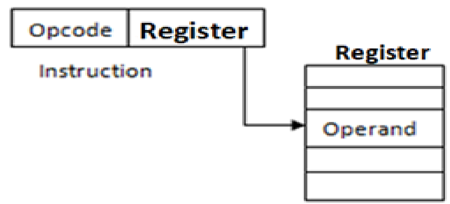


Figure 7.4 Register Direct Addressing Mode

5. **Register Indirect Addressing Mode:** In this mode, the instruction points to a register in the processor whose content contains the address of the operand in the memory. The selected register has the address of the operand instead of having the operand itself. To use this addressing mode, it should be ensured that the memory address of the operand is stored in the register of the processor with the help of prior instruction. For example, LDAX C instruction will load the accumulator with the content of register C. The operand is prior set in register C and then loaded into the accumulator with the help of this instruction.

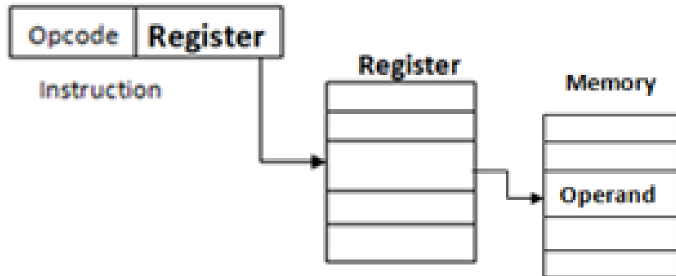


Figure 7.5 Register Indirect Addressing Mode

6. **Relative Addressing Mode:** In the relative mode of addressing, the data of the program counter (PC) is added to the address field of instruction. For instance, if it is desired to find data after 30 bytes from the present content of the PC. This type of addressing mode can be used for a lookup table in designing a subroutine and in conditional jump instructions in the 8051 processor.

7. **Indexed Addressing Mode:** In this mode, the data of an index register (XR) is added to the address field of the instruction. The index register being a special register of the processor contains an index value. If there is no address field in the instruction of indexed addressing mode, then the instruction is automatically executed according to the register indirect addressing mode.

7.6 Types of Instructions

The instruction set of different computers differ from each other mostly in the way the operands are deter-mined from the address and mode fields. The actual operations available in the instruction set are not very different from one computer to another. It so happens that the binary code assignments in the operation code field are differ-ent in different computers, even for the same operation. It may also happen that the symbolic name given to instructions in the assembly language notation is different in different computers, even for the same instruction. Nevertheless, there is a set of basic operations that most, if not all, computers include in their instruction collection. Most computer instructions can be classified into three categories: ***Data Transfer, Data Manipulation,*** and ***Program Control Instructions.***

1. **Data Transfer Instructions**

   The most fundamental type of machine instruction is the data transfer instruction. Data transfer instructions move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves. There are eight data transfer instructions used in many computers, accompanying each instruction is a mnemonic symbol. It must be realized that different computers use different mnemonics for the same instruction name.

   The data transfer instruction must specify several things.

   - The location of the source and destination operands must be

specified. Each location could be memory, a register, or the top of the stack.

- The length of data to be transferred must be indicated.
- As with all instructions with operands, the mode of addressing for each operand must be specified.

Some assembly language conventions modify the mnemonic sym-bol to differentiate between the different addressing modes. For example, the mnemonic for load immediate becomes LDI. Other assembly language conven-tions use a special character to designate the addressing mode. For example, the immediate mode is recognized from a pound sign # placed before the operand. In any case, the important thing is to realize that each instruction can occur with a variety of addressing modes.

2. **Data Manipulation Instructions**

Data manipulation instructions perform operations on data and provide the computational capabilities for the computer. The data manipulation instruc-tions in a typical computer are usually divided into three basic types:

- Arithmetic instructions
- Logical and bit manipulation instructions
- Shift instructions

However, each instruction when executed in the computer must go through the fetch phase to read its binary code value from memory. The operands must also be brought into processor registers according to the rules of the instruction addressing mode. The last step is to execute the instruction in the processor.

Arithmetic Instructions

The four basic arithmetic operations are addition, subtraction, multiplication, and division. Most computers provide instructions for all four operations. Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by means of software subroutines. The four basic arithmetic

operations are sufficient for formulating solutions to scientific problems when expressed in terms of nu-merical analysis methods.

A list of typical arithmetic instructions is given in Table 7.2. The increment instruction adds 1 to the value stored in a register or memory word. One common characteristic of the increment operations when executed in processor registers is that a binary number of all 1' s when incremented produces a result of all 0' s. The decrement instruction subtracts 1 from a value stored in a register or memory word. A number with all D's, when decremented, produces a number with all 1's. An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-preci-sion data.

Logical and Bit Manipulation Instructions

Logical instructions perform binary operations on strings of bits stored in registers. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. The logical instructions consider each bit of the operand separately and treat it as a Boolean variable. By proper application of the logical instructions it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in registers or memory words.

Some typical logical and bit manipulation instructions are listed in Table 7.2. The clear instruction causes the specified operand to be replaced by D's. The complement instruction produces the 1's complement by inverting all the bits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands. Although they perform Boolean operations, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations. There are three bit manipulation operations possible: a selected bit can be cleared to 0, or can be set to 1, or can be complemented.

Shift Instructions

Instructions to shift the content of an operand are quite useful and are

often provided in several variations. Shifts are operations in which the bits of a word are moved to the left or right. The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify either logical shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the right or to the left. Table 7.2 lists four types of shift instructions. The logical shift inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left. Arithmetic shifts usually con-form with the rules for signed-2' s complement numbers.

Table 7.2 Types of Instructions

Instruction Type	**Name**	**Mnemonics**	**Description**
Data Transfer Instructions	Load	LD	Transfer word from memory to processor
	Store	ST	Transfer word from processor to memory
	Move	MOV	Transfer word or block from source to destination
	Exchange	XCH	Swap contents of source and destination
	Input (read)	IN	Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
	Output (write)	OUT	Transfer data from specified source to I/O port or device
	Push	PUSH	Transfer word from source to top of stack

	Pop	POP	Transfer word from top of stack to destination
Data Manipulation Instructions (Arithmetic)	Increment	INC	Add 1 to operand
	Decrement	DEC	Subtract 1 from operand
	Add	ADD	Compute sum of two operands
	Subtract	SUB	Compute difference of two operands
	Multiply	MUL	Compute product of two operands
	Divide	DIV	Compute quotient of two operands
	Add with Carry	ADDC	Compute the sum with carry
	Subtract with borrow	SUBB	Compute the difference with borrow
	Negate (2's complement)	NEG	Change sign of operand
Data Manipulation Instructions (Logical & Bit)	Clear (reset)	CLR	Transfer word of 0s to destination
	NOT	NOT	Perform logical NOT
	AND	AND	Perform logical AND
	OR	OR	Perform logical OR

	Exclusive -OR	XOR	Perform logical XOR
	Enable Interrupt	EI	Enables the Interrupts
	Disable Interrupt	DI	Disables the Interrupts
Data Manipulation Instructions (Shift)	Logical Shift Right	SHR	Right shift operand, introducing constants at end
	Logical Shift Left	SHL	Left shift operand, introducing constants at end
	Rotate Right	ROR	Right shift operand, with wraparound end
	Rotate Left	ROL	Left shift operand, with wraparound end
Program Control Instructions	Branch	BR	Unconditional transfer; load PC with specified address
	Jump	JMP	Jump to specified address
	Skip	SKP	Increment PC to skip next instruction
	Call	CALL	Calling for a subroutine
	Return	RET	Replace contents of PC and other register from known location
	Halt	HLT	Stop Program execution

	Wait (hold)	HLD	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied

3. **Program Control Instructions**

For all of the operation types discussed so far, the next instruction to be performed is the one that immediately follows, in memory, the current instruction. However, a significant fraction of the instructions in any program have as their function changing the sequence of instruction execution. For these instructions, the operation performed by the processor is to update the program counter to contain the address of some instruction in memory.

Instructions are always stored in successive memory locations. When proc-essed in the CPU, the instructions are fetched from consecutive memory locations and executed. Each time an instruction is fetched from memory, the program counter is incremented so that it contains the address of the next instruction in sequence. After the execution of a data transfer or data manip-ulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence.

On the other hand, a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered. In other words, program control instructions specify conditions for altering the content of the program counter, while data transfer and manipulation in-structions specify conditions for data-processing operations. The change in value of the program counter as a result of the execution of a program con-trol instruction causes a break in the sequence of instruction execution. Some typical program control instructions are listed in Table 7.2.

Branch and Jump Instructions: The branch and jump instructions are used interchangeably to mean the same thing, but sometimes they are

used to denote different addressing modes. The branch is usually a one-address instruction. It is written in assembly language as BR ADR, where ADR is a symbolic name for an address. When executed, the branch instruction causes a transfer of the value of ADR into the program counter. Since the program counter contains the address of the instruction to be executed, the next instruction will come from location ADR. Branch and jump instructions may be ***conditional*** or ***unconditional***. An unconditional branch instruction causes a branch to the specified address without any conditions. The conditional branch instruction specifies a condition such as branch if positive or branch if zero. If the condition is met, the program counter is loaded with the branch address and the next instruction is taken from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.

Skip Instructions: The skip instruction does not need an address field and is therefore a zero-address instruction. A conditional skip instruction will skip the next instruction if the condition is met. This is accomplished by incrementing the program counter during the execute phase in addition to its being incremented during the fetch phase. If the condition is not met, control proceeds with the next instruction in sequence where the programmer inserts an unconditional branch instruction. Thus a skip-branch pair of instructions causes a branch if the condition is not met, while a single conditional branch instruction causes a branch if the condition is met.

7.7 Reduced Instruction Set Computer (RISC)

A set of instructions is executed by every processor. As the number of complex instructions within the instruction set of any processor increases, the instruction decoding becomes more complex and time-consuming. In the late 1970s and early 1980s, the aim of the processor was to incorporate more and more complex instruction sets. Furthermore, an added burden to the

architectural design of such processors was to maintain backward compatibility with the previous processors. After a considerable amount of analysis and research, it was understood that the simpler instructions are executed more times than their complex counterparts. The second point that came up is the relative speed of the processor and memory. Although both the complexity of instructions and the speed are tending towards improvement, still the memory is a slower device to communicate, in comparison to the execution speed of the processor. The designers categorize these processors as **RISC** (Reduced instruction set computing) and **CISC** (Complex instruction set computing).

RISC Processors

RISC processor was introduced in Berkeley in the year 1980, a processor with a reduced number of instruction sets, avoiding all complex instructions. Simpler instructions consume less time to be executed. Furthermore, there was a limited memory for loading and storing data instructions only. In other words, the instructions like *'increment the content of a memory location by one'* are avoided. To reduce memory accessing time, a larger number of registers were provided within the processor itself. Finally, the pipeline architecture method was adopted to speed up the execution of instruction.

RISC processors have the following special characteristics.

- The reduced and restricted number of instructions.
- Simpler instructions, avoiding complex instructions.
- Lesser number of addressing modes.
- Simple and uniform instruction format so that most instructions may be executed within one machine cycle.
- A larger number of registers within the processor to reduce external memory access time.
- Pipeline architecture.

Schematically, the difference between RISC and CISC is illustrated in Table 5.2. It may be indicated that the architecture of 8051 microcontrollers is closer to RISC architecture, while the architecture adopted for 8086 might be

designated closer to CISC. it should be thoroughly noted that CISC processors can not adopt the pipeline architecture or some other features of RISC. The basic objective of the RISC architecture is to speed up the working of the processor even with the same clock speed.

Table 5.2 Difference between RISC and CISC architecture

RISC Architecture	CISC Architecture
Reduced number of instructions.	A larger number of instructions.
Simpler and straight forward addressing modes.	Complex and extensive addressing modes.
More internal registers.	Limited internal registers.
Simpler instruction format.	Complex instruction format.
Pipeline architecture.	Non- pipeline architecture.
Most instructions are executed within one machine cycle.	Many instructions consume multiple machine cycles.

General Features of RISC processors:

- **Simple and Reduced Number of Instructions:** As the complexity of decoding any instruction of any processor increases with the number of instructions to be decoded by it. The simpler structure of the instruction decoder enhances the performance of any processor. Therefore, in RISC processors, the number of instructions is restricted. Even the multiply and the divide instructions are considered as complex instructions and the classical RISC architecture allows add and subtract instructions only.
- **Lesser Addressing Modes:** Addressing mode represents the way in which a target data can be located. However, more addressing modes can also make the instruction decoding complicated.
- **Uniform Instruction Format:** It should be pointed out that non-uniformity in the instruction length (number of bytes) forces the

processor for poor performance. If the length of the instruction is uniform then it is easier for the processor to allot that much memory to the instruction and take further action accordingly.

- **More Registers:** As indicated before, a memory oriented processor is to constantly access external memory even for simpler operations, e.g., add two numbers and store. This external memory access is more time-consuming than having all necessary operands within the processor itself, in its register set. The same set of internal registers also may be used to store the results. Although the optimum number of CPU registers is still a matter of debate, most RISC architecture offers 32 or more internal registers within the processor. On the other hand, a CISC processor would offer about six to eight internal registers for the processor's operand and result in storage purposes.
- **Pipeline Architecture:** The pre-fetching instruction bytes during decoding and execution of ongoing instructions speeds up the processing. Further efficiency might be achieved by concurrent fetching, decoding, and operand fetching, execution, and result in storage operations. Therefore, it is essential for a RISC processor to adopt the pipeline architecture. It may also be adopted by most CISC processors.

7.8 Summary

- The part of the computer that performs the bulk of data-processing operations is called the Central Processing Unit (CPU).
- A group of flip-flops form a register. A register is a special high speed storage area in the CPU.
- Registers perform two important functions in the CPU operation. First, providing a temporary storage area for data. This helps the currently executing programs to have a quick access to the data, if needed. Second, storing the status of the CPU as well as information about the currently executing program.

- Control Word consists of four fields. Three fields contain three bits each, and one field has five bits.
- Subroutines are the pre-defined functions that can be called whenever required.
- Addressing modes display the method by which the data is targeted by the instruction.
- Most computer instructions can be classified into three categories: ***Data Transfer, Data Manipulation,*** and ***Program Control Instructions.***

7.9 Key Terms

- **Stack-top:** The particular location within the stack is known as the ***stack-top.***
- **Stack Pointer:** The address of the stack top is available in the register designated for the specific purpose and hence known as the ***stack pointer.***
- **RISC:** A reduced instruction set computer, or RISC, is a computer with a small, highly optimized set of instructions.
- **CISC:** A complex instruction set computer, or CISC, is a computer in which single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions.
- **Transistors:** Transistors are microscopic bits of material that block electricity at one voltage (non-conductor) and allow electricity to pass through them at different voltage (conductor).

7.10 Check Your Progress

Q1) Give a broad classification of Instructions.

Q2) Differentiate between RISC and CISC.

Q3) Define Control Word. What is the role of Control Word in the General Register Organization?

Q4) Discuss the most widely used addressing modes briefly.

Q5) Write a short note on Program Control Instructions.

References

Computer System Architecture, M. Morris Mano

Computer Organization and Architecture, 9th edition, William Stallings, Pearson Publication.

Computer Architecture and Organization, Subrata Ghoshal, Pearson Publication.

Unit 8 – The Memory Systems

Structure

8.0 Introduction
8.1 Unit Objectives
8.2 Memory Classification
 8.2.1 Read Only Memory (ROM) 8.2.2 Read/ Write Memory (RAM)
8.3 Memory Characteristics and Hierarchy
 8.3.1 Cache Memory 8.3.2 Main Memory
 8.3.3 Secondary Memory 8.3.4 Virtual Memory
8.4 Memory Management
8.5 Memory Decoding
8.6 Summary
8.7 Key Terms
8.8 Check Your Progress

8.0 Introduction

The system memory is where the computer holds current projects and information that are being used. There are different degrees of computer memory, including ROM, RAM, store, page, and illustrations, each with explicit destinations for system activity. This area reflects around the job of computer memory, and the innovation behind it".

In spite of the fact that memory is used in a wide range of structures around present-day computer systems, it tends to be partitioned into two basic sorts: RAM and ROM. ROM, or Read-Only Memory, is generally little, however basic to how a computer functions. ROM is constantly found on motherboards, yet is progressively found on designs cards and some other development cards and peripherals. As a rule, ROM doesn't change. It shapes the essential guidance set for working the equipment in the system, and the information inside stays flawless in any event when the computer is closed down. It is conceivable to

refresh ROM, yet it's just done when required. If the ROM is harmed, the computer system essentially can't work".

The present-day computers have altogether more memory than the main computers of the mid 1980s, and this has affected the improvement of the computer's design. The difficulty is, putting away and recovering information from an enormous square of memory is additional tedious than from a little square. With a lot of memory, the distinction in time between a register and a memory is incredible, and this has brought about additional layers of reserve in the capacity order".

8.1 Unit Objectives

This unit will help the reader to gain knowledge about:

- General organization of memory system in computers.
- Classification of Memory and their applications.
- Basic features of DMA (Direct Memory Access).
- Concept of Memory Decoding.

8.2 Memory Classification

Memory is the most fundamental component of an operating system without which computers can't perform straight and simple tasks. Computer memory is of two essential sort – Primary memory (RAM and ROM) and Secondary memory (hard drive, CD, etc.). Random Access Memory (RAM) is an essential volatile memory and Read-Only Memory (ROM) is an essential non-volatile memory.

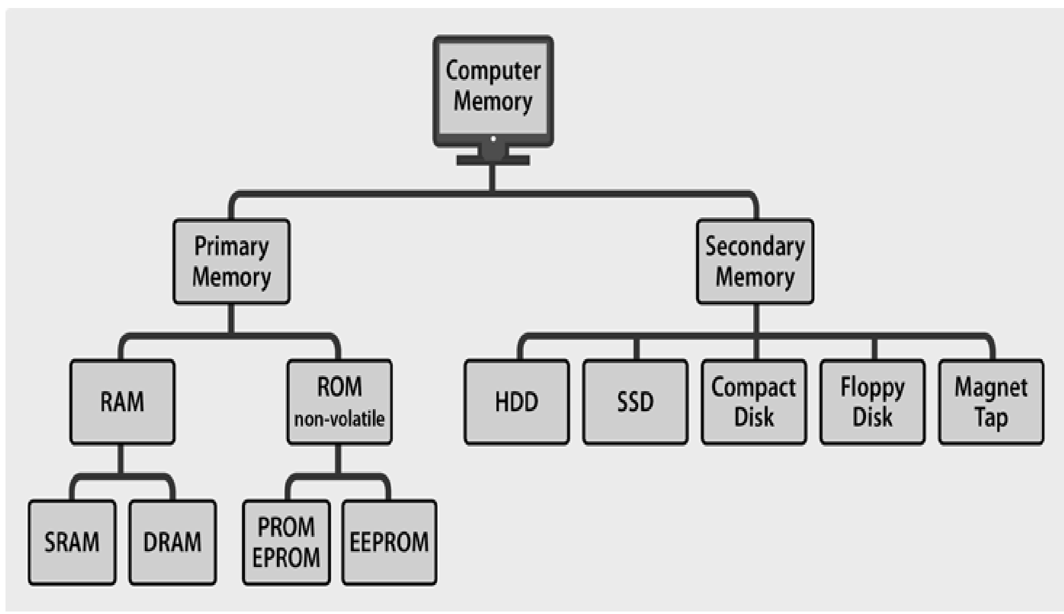


Figure 8.1 Memory classification

(**Source-** https://www.enterprisestorageforum.com/storage-hardware/types-of-computer-memory.html)

8.2.1 Read Only Memory (ROM)

ROM is an abbreviation for Read-Only Memory. It refers to computer memory chips containing lasting or semi-perpetual information. In contrast to RAM, ROM is non-volatile; significantly after you turn off your computer, the substance of ROM will regain. Pretty much every computer accompanies a modest quantity of ROM containing the boot firmware. This comprises of a couple of kilobytes of code that guide the computer when it fires up, e.g., running equipment diagnostics and stacking the working system into RAM. On a computer, the boot firmware is known as the **BIOS**”. To refresh ROM, different methods and technologies are being used. Rewritable ROM chips incorporate PROMs (programmable read-just memory), EPROMs (erasable read-only memory), EEPROMs (electrically erasable programmable read-only memory), and a typical variety of EEPROMs called streak memory. In any case, these sorts of non-volatile memory can be adjusted and are regularly alluded to as programmable ROM.

Since ROM can't be changed, it is basically used for the firmware. Firmware is programming projects or sets of directions that are installed into an equipment gadget. It supplies the required guidelines on how a gadget speaks with different equipment segments. Firmware is avoided as semi-perpetual due to the fact that it doesn't change except if it is refreshed. Firmware incorporates BIOS, erasable programmable ROM (EPROM), and the ROM arrangements for programming.

- **Programmable Read-Only Memory (PROM):** It is a programmable read-only memory that can be programmed once by the user according to the need and the data remains permanent in PROM. It is a non-volatile memory.
- **Erasable Programmable Read-Only Memory (EPROM):** Erasable programmable read-only memory is a type of ROM that can be erased and reused, unlike PROM. The memory is erased using UV-rays. The EPROM chip has to be removed from the system and then erased and reprogrammed. This is modified with the usage of extremely high voltages and introduction to roughly 20 minutes of exceptional bright (UV) light.
- **Electrically-Erasable Programmable Read-Only Memory (EEPROM):** Electrically erasable programmable read-only memory can be erased and reprogrammed repeatedly by applying a higher voltage pulse (about 5V DC) for milliseconds. There is no need to remove the chip each time, it is user-modifiable. It is also termed as an upgraded version of EPROM. This is used in numerous more established computer BIOS chips that can be deleted and modified a few times and permits just a single area at once to be composed or eradicated. A refreshed variant of EEPROM is streak memory; this permits various memory areas to be adjusted at the same time.
- **FLASH:** It is similar to EEPROM with a minute difference. In EEPROM, the data is erased byte-wise while Flash removes the whole data at one time.

ROM is likewise regularly used in optical capacity media, for example, different sorts of minimal plates, including read-just memory (CD-ROM), conservative circle recordable (CD-R), and reduced circle rewritable (CD-RW).

8.2.2 Read/Write Memory (RAM)

RAM (Random Access Memory) is the equipment in a figuring gadget where the operating system (OS), application projects, and information in current use are kept so they can be immediately reached by the gadget's processor. RAM is the primary memory in a computer, and it is a lot quicker to peruse from and write to than different sorts of capacity, for example, a hard disk drive (HDD), or optical drive.

Random-access Memory is volatile. That implies information is held in RAM as long as the computer is on, however it is lost when the computer is off. At the point when the computer is rebooted, the OS and different records are reloaded into RAM, as a rule from an HDD or SSD. As a result of its instability, RAM can't store lasting information. Initially, the term Random Access Memory was used to recognize ordinary center memory from offline memory. The offline memory is suggested to magnetic tape from which a particular bit of information must be obtained by finding the location successively, beginning toward the start of the tape.

8.3 Memory Characteristics and Hierarchy

The key characteristics of the memory system are as follows:

- Location
- Capacity
- Unit of Transfer
- Access Method
- Performance
- Physical Type
- Physical Characteristics

- Organization

1. **Location**: It manages the area of the memory device in the computer system. There are three potential areas:
   - CPU: This is frequently termed as CPU registers and a limited quantity of storage.
   - Interior or primary: This is the fundamental memory like RAM or ROM. The CPU can straightforwardly get to the primary memory.
   - Outside or auxiliary: It includes optional capacity devices like hard disks, magnetic tapes. The CPU doesn't get to these devices legitimately. It utilizes gadget controllers to get to auxiliary stockpiling devices.
2. **Capacity**: The limit of any memory gadget is communicated as far as i) word size ii) Number of words
   - Word size: Words are communicated in bytes (8 bits). A word can anyway mean any number of bytes. Usually used word sizes are 1 byte (8 bits), 2bytes (16 bits), and 4 bytes (32 bits).
   - A number of words: This determines the number of words accessible in the specific memory gadget. For instance, if a memory gadget is given as 4K x 16. This means the device has a word size of 16 bits and a sum of 4096(4K) words in memory.
3. **Unit of Transfer:** It is the greatest number of bits that can be perused or composed into the memory at once. If there should be an occurrence of primary memory, it is for the most part equivalent to a word size. If there should arise an occurrence of outside memory, the unit of the move isn't restricted to the word size; it is regularly bigger and is alluded to as locations.
4. **Access Methods:** It is a basic quality of memory devices. It is the succession or request wherein memory can be obtained. There are three kinds of access strategies:
   - Random Access: If capacity areas in a specific memory gadget can be achieved in any request and access time is free of the memory area

being. Such memory devices are said to have an arbitrary access component. Slam (Random Access Memory) IC's usage of this entrance strategy.

- Serial Access: If memory areas can be obtained to just in a specific foreordained succession, this entrance strategy is called sequential access. Magnetic Tapes, CD-ROMs utilize sequential access techniques.
- Semi-random Access: Memory devices, for example, Magnetic Hard plates utilize this entrance strategy. Here each track has a perused/compose head in this manner each track can be received to randomly access inside each track is confined to sequential access.

5. **Performance:** The display of the memory system is resolved to utilize the following three parameters.
    - Access Time: In random access memories, it is the time taken by memory to finish the read/compose activity from the moment that a location is sent to the memory. For non-random access memories, it is the time taken to situate the read composes head at the ideal area. Access time is broadly used to quantify the execution of memory devices.
    - Memory cycle time: It is characterized specifically for Random Access Memories and is the total of the appearance time and the extra time required before the subsequent access can begin.
    - Transfer rate: It is characterized as the rate at which information can be moved into or out of a memory unit.
6. **Physical Type:** Memory devices can be either semiconductor memory (like RAM) or magnetic surface memory (like hard drives).
7. **Physical Characteristics:**
    - Volatile/Non-Volatile: If a memory device proceeds with, hold data regardless of whether power is off. The memory device is non-volatile else it is volatile.

8. **Organization:**
    - Erasable/Non-erasable: The memories where information once customized can't be eradicated are called Non-erasable memories. Memory devices in which information in the memory can be deleted is called erasable memory. Example RAM (erasable), ROM (non-erasable).

8.3.1 Cache Memory

Cache Memory is an exceptional fast memory. It is used to accelerate and synchronize with a fast CPU. Secondary memory is costlier than principle memory yet practical than CPU registers. It holds as often as possible mentioned information and directions with the goal that they are promptly accessible to the CPU when required. Cache memory is utilized to lessen the normal opportunity to get to information from the Main memory. There are different diverse free reserves in a CPU, which store guidelines and information".

Cache Performance

"At the point when the processor needs to peruse or compose an area in fundamental memory, it first checks for a relating section in the reserve. On the off chance that the processor finds that the memory area is in the reserve, a store hit has happened and information is perused from the store". "On the off chance that the processor doesn't discover the memory area in the reserve, a store miss has happened. For a store miss, the reserve dispenses another passage and duplicates information from primary memory, at that point the solicitation is satisfied with the substance of the store. The presentation of store memory is much of the time estimated regarding an amount called the Hit proportion". We can improve Cache execution utilizing higher cache block size, higher associability, decrease miss rate, lessen miss punishment, and diminish Reduce an opportunity to hit in the reserve.

Application of Cache Memory

"As indicated, the cache memory can store a sensible number of locations at a time, however, this number is little contrasted with the number of locations in the primary memory. The correspondence between the primary memory locations and those in the store is determined by a mapping capacity".

Types of Cache

- **Primary Cache –** A primary cache unit, denoted as Level 1 (L1), is situated on the processor chip itself. This memory is fast but small and its entrance time is practically identical to that of processor registers.
- **Secondary Cache –** The Secondary cache unit exists between the primary storage and the processor. It is indicated as Level 2 (L2) cache storage. It is larger in size than L1 and is connected externally to the processor chip.

Mapping of Cache Memory

The basic characteristic of cache memory is its fast access time. Therefore, very little or no time must be wasted when searching for words in the cache. The transformation of data from main memory to cache memory is referred to as a ***mapping process***. Three types of mapping procedures are of practical interest when considering the organization of cache memory:

➔ ***Associative Mapping:*** The fastest and most flexible cache organization uses an associative memory. The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory. For example, the address value of 15 bits is written as a five-digit octal number and its corresponding 12 -bit word is written as a four-digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12-bit data is read and sent to the CPU. If no match occurs, the main memory is accessed for the word. The ad-dress-data pair is then transferred to the associative cache memory. If the cache is full, an address-data pair must be displaced to make room for a pair that

is needed and not presently in the cache. The decision as to what pair is replaced is determined from the replacement algorithm that the designer chooses for the cache. A simple procedure is to replace cells of the cache in round-robin order whenever a new word is requested from main memory. This constitutes a first-in first-out (FIFO) replacement policy.

➔ ***Direct Mapping:*** Associative memories are expensive compared to random-access memories because of the added logic associated with each cell. The possibility of using a random-access memory for the cache. The CPU address of 15 bits is divided into two fields. The nine least significant bits constitute the index field and the remaining six bits form the tag field. The main memory needs an address that includes both the tag and the index bits. The number of bits in the index field is equal to the number of address bits required to access the cache memory. In the general case, there are 2^k words in cache memory and 2^n words in main memory. The n-bit memory address is divided into two fields: k bits for the index field and n - k bits for the tag field. The direct mapping cache organization uses the n-bit address to access the main memory and the k-bit index to access the cache. In the internal organization of the words in the cache memory, each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache. The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value. The disadvantage of direct mapping is that the hit ratio can drop considerably if two or more words whose addresses have the same index but different tags are accessed repeatedly. However, this possibility is

minimized by the fact that such words are relatively far apart in the address range.

- ➔ ***Set-associative Mapping:*** A third type of cache organ-ization, called set-associative mapping, is an improvement over the direct- mapping organization in that each word of cache can store two or more words of memory under the same index address. Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set. In general, a set-associative cache of set size k will accommo-date k words of main memory in each word of cache.

Writing into Cache Memory

An important aspect of cache organization is concerned with memory write requests. When the CPU finds a word in cache during a read operation, the main memory is not involved in the transfer. However, if the operation is a write, there are two ways that the system can proceed.

Write-through Method

The simplest and most commonly used procedure to update main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is called the ***write-through method.*** This method has the advantage that the main memory always contains the same data as the cache. This characteristic is important in systems with direct memory access transfers. It ensures that the data residing in main memory are valid at all times so that an I/O device communicating through DMA would receive the most recent updated data.

Write-back Method

The second procedure is called the ***write-back method***. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the word is removed from the cache it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times; however, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the

cache. It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory. Analytical results indicate that the number of memory writes in a typical program ranges between 10 and 30 percent of the total references to memory.

Cache Initialization

One more aspect of cache organization that must be taken into consideration is the problem of initialization. The cache is initialized when power is applied to the computer or when the main memory is loaded with a complete set of programs from auxiliary memory. After initialization the cache is considered to be empty, but in effect it contains some nonvalid data. It is customary to include with each word in cache a ***valid bit*** to indicate whether or not the word contains valid data.

The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache word is set to 1 the first time this word is loaded from main memory and stays set unless the cache has to be initialized again. The intro-duction of the valid bit means that a word in cache is not replaced by another word unless the valid bit is set to 1 and a mismatch of tags occurs. If the valid bit happens to be 0, the new word automatically replaces the invalid data. Thus the initialization condition has the effect of forcing misses from the cache until it fills with valid data.

Cache Coherence

In contemporary multiprocessor systems, it is customary to have one or two levels of cache associated with each processor. This organization is essential to achieve reasonable performance. It does, however, create a problem known as the ***cache coherence problem***. The essence of the problem is this: Multiple copies of the same data can exist in different caches simultaneously, and if processors are allowed to update their own copies freely, an inconsistent view of memory can result.

For any cache coherence protocol, the objective is to let recently used local variables get into the appropriate cache and stay there through numerous reads and writes, while using the protocol to maintain consistency of shared

variables that might be in multiple caches at the same time. Cache coherence approaches have generally been divided into software and hardware approaches. Some implementations adopt a strategy that involves both software and hardware elements. Nevertheless, the classification into software and hardware approaches is still instructive and is commonly used in surveying cache coherence strategies.

8.3.2 Main Memory

The main memory of a computer is called Random Access Memory. It is otherwise called RAM. This is the piece of the computer that stores working framework programming, programming applications, and other data for the focal handling unit (CPU) to have quick and direct access when expected to perform errands. It is classified "irregular access" due to the fact that the CPU can go straightforwardly to any area of fundamental memory, and doesn't have to approach the procedure in a successive request".

Process: The focal handling unit is one of the most significant parts of the computer. It is the place different undertakings are performed and output is created. At the point when the chip finishes the execution of a lot of directions and is prepared to complete the following assignment, it recovers the data it needs from RAM. Commonly, the bearings incorporate the location where the data, which should be perused, is found. The CPU transmits the location to the RAM's controller, which experiences the way towards finding the location and perusing the information".

Dynamic RAM (DRAM)

"Dynamic random access memory (DRAM) is the most well-known sort of fundamental memory in a computer. It is a pervasive memory source in computers, just as workstations. DRAM is continually re-establishing whatever data is being held in memory. DRAM stores each bit of data in a separate capacitor, thus require more power than SRAM.

Static RAM (SRAM)

"Static Random Access Memory (SRAM) is the second kind of principal memory in a computer. Information held in SRAM doesn't need to be constantly invigorated; data in this fundamental memory stays as a "static picture" until it is overwritten or is erased when the power is turned off. Since SRAM is not so thick but rather more power productive when it isn't being used; in this way, it is a superior decision than DRAM for specific uses like memory locations situated in CPUs. It is costlier than DRAM.

Synchronous RAM (SDRAM)

SDRAM is the most widely used type of DRAM. The traditional DRAM is asynchronous in nature i.e. does not depend upon the clock cycle of the system. SDRAM utilizes an external clock signal for synchronous coordination with the processor. It comprises a mode register and associated control logic so that it can be customized according to the specific system needs. It is also capable of accessing bulk data at a time and is used for worksheets and multimedia files.

Adequate RAM

The CPU is frequently viewed as the most significant component in the exhibition of a computer. Smash most likely arrives in a nearby second. Having a satisfactory measure of RAM directly affects the speed of the computer. A framework that needs enough fundamental memory to run its applications must depend on the working framework to make extra memory assets from the hard drive by "trading" information in and out. At the point when the CPU must recover information from the circle rather than RAM, it hinders the exhibition of the computer. Numerous games, video-altering or design programs require a lot of memory to work at an ideal level".

8.3.3 Secondary Memory

"Primary memory has a restricted capacity limit and is unpredictable. Secondary memory is also known as Auxiliary memory, overcomes this constraint by giving perpetual storing of information in mass amounts.

Auxiliary memory is likewise named as outside memory and alludes to the different storage media on which a computer can store information and projects. Secondary storage media can be fixed or removed. Fixed Storage media is an inner storage medium like a hard disk that is fixed inside the computer. The storage medium that is convenient and can be taken outside the computer is named as removable storage media".

Advantages of Secondary storage:

- **Changeless Storage:** Primary Memory (RAM) is unpredictable, for example, it loses all data when the power is off, so as to make sure about the information for all time in the system, and secondary storage devices are required.
- **Compactness:** Storage medium, similar to the CDs, streak drives can be utilized to move the information starting with one device then onto the next.

Fixed and Removable Storage

- **Fixed Storage-** A Fixed storage is an inside media device that is utilized by a computer framework to store information, and for the most part these are alluded to as the Fixed Disks drives or the Hard Drives". "Fixed storage devices are truly not fixed, clearly these can be expelled from the framework for fixing work, upkeep reason, and furthermore for update and so on.
- **Removable Storage-** "This is an external media device that is used by a computer context to store data, and for the most part, these are mentioned as the Removable Disks drives or the External Drives. Removable storage is any sort of capacity gadget that can be launched out from a computer framework while the framework is running. Instances of outer gadgets incorporate CDs, DVDs, and Blu-ray disc drives, just as diskettes and USB drives. Removable capacity makes it simpler for a client to move information starting with one computer framework then onto the next". The kinds of Removable Storage are:

- Optical disks (CDs, DVDs, Blu-ray discs)
- Memory cards
- Floppy disks
- Magnetic tapes
- Removable Hard disk drive (HDD)

8.3.4 Virtual Memory

Virtual memory is an element of an operating system that empowers a computer to be able to compensate for lack of physical memory by moving pages of data from random access memory to disk storage. This procedure is done temporarily and is intended to work as a combination of RAM and space on the hard disk. This implies that when RAM runs low, virtual memory can move data from it to a space called a ***paging file***. This process further permits RAM to be freed up so that a computer can complete the task.

Some of the characteristics of virtual memory are discussed below:

- **Physical and Virtual Addresses**

  A computer contacts the contents of its RAM through a system of addresses, which are basically numbers that find each byte. Since the amount of memory differs from computer to computer, defining which software will function on a computer becomes difficult. Virtual memory takes care of this issue by treating every computer as if it has a lot of RAM and each program as if it uses the computer totally. The operating system, for example, Microsoft Windows or Apple's OS X, makes a set of virtual addresses for every program. The OS decodes virtual addresses into physical ones, vigorously fitting programs into RAM as it becomes available.

- **Paging in Virtual Memory**

  Virtual memory breaks down the programs into fixed-size blocks called ***pages***. If a computer has a rich physical memory, the operating system loads the entire program's pages into RAM. If not, the OS fits as much as it can and runs the instructions into those pages. When the computer

finishes its work with those pages, it loads the remaining part of the program into RAM, by overwriting previous pages. Because the operating system spontaneously achieves these details, this liberates the software developer to focus on program features and not stress about memory problems.

- **Multiprogramming**

  Virtual memory with paging lets a computer run several programs simultaneously, regardless of available RAM. This benefit, called multiprogramming, is a key component of modern computer operating systems, as they accommodate various utility programs like printer drivers, virus scanners, and network managers at the same time as the applications - Web browsers, word processors, E-mail and media players.

- **Paging File**

  With virtual memory, the computer composes program pages that have not been used recently to a part on the hard drive called a paging file. The file keeps the data contained in the pages; if it is needed again by the program, the operating system reloads it when RAM. When several programs strive for RAM, the process of swapping pages to the file can slow a computer's handling speed, as it devotes more time in doing memory management chores and less time completing useful work. Ideally, a computer will have enough RAM to handle the requests of many programs, reducing the time the computer spends dealing with its pages.

8.4 Memory Management

The memory system needs suitable attention to be properly utilized, hence, this technique is called memory management. Within the operating system, the memory system is composed of software and hardware. There are some duties to be performed by the memory manager:

- It has to shield the memory area from unwanted access.

- Use the accessible memory area in the best mode possible for all relevant software.
- It has to maintain the track of the whole memory space, both occupied as well as free.
- It has to assign spaces as and when needed to the respective programs.
- It has to create addresses as per the system necessities.
- Also, to supervise the smooth running of the system in memory usage.

Direct Memory Access (DMA)

DMA basically stands for direct memory access. There is a hardware device which is used for direct memory access is known as the DMA controller. A DMA controller is a control unit, part of the I/O device's interface circuit, which can transfer blocks of data/information between I/O devices and main memory.

It needs the synchronous working of more than one section of a computer for the completion of a computer program. The most important thing is in what way it handles the transfer of data among memory, processor and I/O devices. Generally, processors control the entire process of transferring information, right from beginning the transfer to the data storage at the destination. This increases the load on the processor. Most of the time, it remains in the ideal state, thus reducing the effectiveness of the system. To increase the speed of data transfer between I/O devices and memory, the DMA controller acts as a station master. Now we will discuss how DMA transmits the data and briefly about its block diagram, in the section below.

DMA controller comprises an address unit, for producing addresses and selecting an I/O device for transfer. It also consists of the control unit and the data count. These are there for keeping counts of the number of blocks transferred and signifying the direction of transfer of data. When the transfer is

finished, DMA informs the processor by raising an error. The typical block diagram of the DMA controller is shown in the figure below.

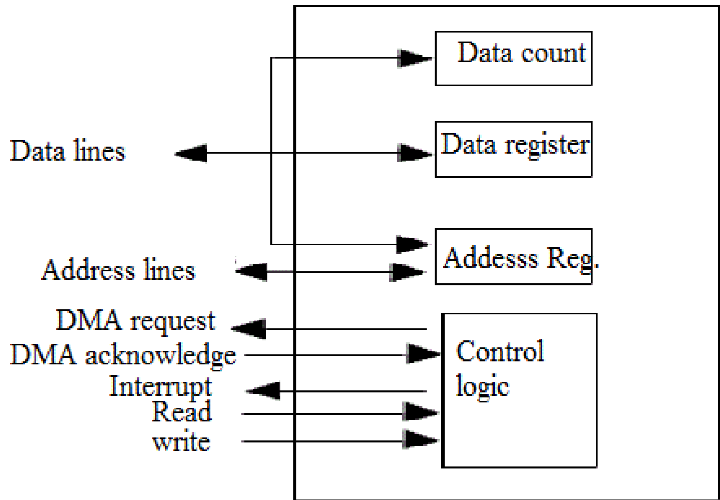


Figure 8.2 Typical Block Diagram of the DMA Controller

(Source- https://www.elprocus.com/direct-memory-access-dma-in-computer-architecture/)

Data Transfer by DMA Controller

The DMA controller transfers the data according to the instructions received by the processor. After completion of the transfer of data, it restricts the bus request signal, and also the CPU disables the bus grant signal. This causes the moving of control buses to the CPU.

Whenever an I/O device needs to initiate the transfer of data then it automatically sends a DMA request signal to the DMA controller, after which the controller accepts if it is free. With raising the bus request signal, the controller appeals for the bus. The data is transferred from the device after receiving the bus grant signal. In this controller, *n* external devices can be connected to it for *n* channeled DMA controller.

The transfer of data takes place in three modes which are stated as follows:

a. **Burst Mode**: In this mode, DMA provides the buses to CPU only after the accomplishment of the entire data transfer. In the meantime, if the CPU needs the bus it has to stay ideal and pause until the data transfers.

b. **Cycle Stealing Mode**: In this mode, DMA gives the entire control of buses to CPU after the transfer of each and every byte. Now, continuously, it issues a request for bus control, makes the transfer of one single byte and returns the bus. If it wants a bus for a higher priority task, the CPU does not wait for a long time due to this mode.

c. **Transparent Mode:** in this mode, when CPU is completing the instruction, DMA transfers only data and it does not need the use of buses.

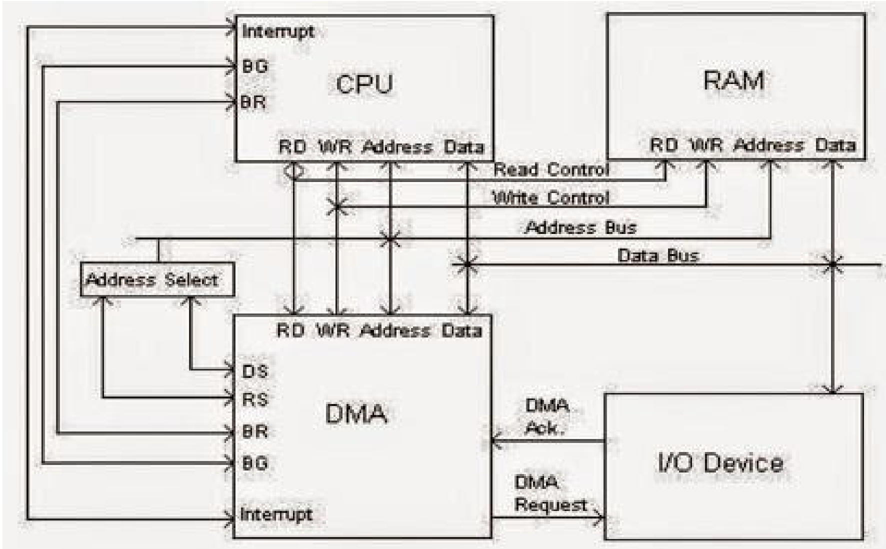


Figure 8.3 Transfer of data in the computer by DMA Controller

(Source- https://www.elprocus.com/direct-memory-access-dma-in-computer-architecture/)

8.5 Memory Decoding

Whenever a processor makes an external address with a usable memory read control signal, the corresponding memory site keeps the data in the data bus to drive it to the processor. This is apparently achieved by a module which is called a memory decoder.

Generally, the primary memory area of any of the processors is huge in size, say, as big as 1GB or maybe more. It has to keep in mind that several memory chips are located and interconnected to allow that much of memory area to the processor. It is very rare when one memory device provides the whole memory

area. This is certainly because even if one of the memory devices starts malfunctioning, there are other active memory devices that continue with their responsibilities. If the whole of the memory area is controlled by only one memory chip of 1GB, then any kind of malfunctioning of that particular chip stops the whole system from functioning.

Types of Decoder

There are three general-purpose decoders. They are:

- **74139 dual 2-to-4 decoder (16-pin DIP):** This decoder offers two 2-to-4 decoders in a 16 pin DIP which means dual in-line package. It is perfect for a small system with a maximum of four memory and four input-output devices.
- **74138 3-to-8 decoder (16-pin DIP):** This type of decoder is adopted for a medium system. Say, up to eight memory devices 3-to-8 decoder in 16-pin DIP.
- **74159 4-to-16 decoder (24-pin DIP):** This type of decoder is adopted for a large system which is up to sixteen memory devices 4-to-16 decoder in 24-pin DIP.

Please note that, in the systems used for these decoders, the whole of the addressable memory area is decoded, but only a section comprises memory devices of this area and the rest of the other section is left for future extension purposes.

The addressable memory area is distributed into several equivalent sizes and every part of this is engaged with a memory device separately. Therefore, the number of separations depends upon the total addressable memory area and the accessible size of memory devices. For example, if 64kB is the total addressable memory space for a processor (e.g., 80885) and if 8kB memory devices are available, then a total of eight such memory devices would be necessary to make the complete memory system, which in turn would need a 3-to-8 decoder, i.e., 74138. The reader may note that general practice is to

divide the available memory area into 2^n segments or parts for obvious reasons.

Common issues: Due to various factors, the system designer in some cases might not have other options but to divide the memory area into unequal parts. This may arise because the memory might be composed of RAM and ROM, and ICs of equal sizes might not be available. Whatever be the reason, the decoding problem may be solved by adequate planning and usage of extra hardware in some cases.

8.6 Summary

- The memory of the computer is broadly categorized as primary and secondary memory. The Primary memory stores all the programs and software of the system while secondary memory provides large and external storage to the system. Primary memory is further classified as RAM (Random Access Memory) and ROM (Read-only memory). RAM is understood as a volatile memory while ROM is a non- volatile memory.
- SRAM (Static RAM), DRAM (Dynamic RAM), and SDRAM (Synchronous DRAM) are the types of RAM and PROM (Programmable ROM), EPROM (Erasable PROM), and EEPROM (Electronically EPROM) are types of ROM.
- Cache Memory holds the frequently accessed and requested data so that it is not necessary to access data from the main memory. This reduces the average time of data access. It is extremely fast and acts as a buffer between the CPU and RAM. According to the need, the cache memory is subdivided into L1 and L2 types.
- Virtual memory is an element of an operating system that empowers a computer to be able to compensate for lack of physical memory by moving pages of data from random access memory to disk storage.
- A DMA controller is a control unit, part of the I/O device's interface circuit, which can transfer blocks of data/information between I/O devices and main memory.

8.7 Key Terms

- **Firmware:** It is a software program that contains the instructions for how the device connects with the system hardware. It is stored in the ROM of the computer. There is a need to update the firmware for better performance.
- **BIOS:** BIOS is a Basic Input- Output System that is primarily used in booting up of the operating system and system set up. It also manages the data flow between the hardware peripherals and the operating system. It is also stored in the ROM.
- **Blu- ray disc:** It is an optical disc used for displaying high- quality videos and can store a large amount of data (more than a DVD).
- **Hard Disk Drive:** It is a high capacity storage device ranging up to terabytes. It can be removable or non- removable in the system. It comprises sectors and tracks; the data is stored in these magnetic tracks and is accessed by the read/ write head.

8.8 Check Your Progress

Q1) State the differences between:

a) RAM and ROM

b) EPROM and EEPROM

c) SRAM and DRAM

Q2) Explain the concept of Memory decoding in detail.

Q3) Write a short note on:

a) DMA controllers

b) Virtual Memory

c) Cache Memory

Q4) Describe the main characteristics of the memory system in detail.

Q5) What are the four characteristics of Virtual Memory and how it works?

Q6) What is the advantage of using cache memory at different levels?

Q7) Give the classification of memory in the computer system.

References:

Computer Architecture and Organization, Subrata Ghoshal, Pearson Publication.

Computer Organization and Architecture, *9th edition*, William Stallings, Pearson Publication.

Computer System Architecture, M. Morris Mano

https://www.sciencedirect.com/topics/computer-science/cache-memory

Unit 9 – Control Unit

Structure

9.0 Introduction
9.1 Unit Objectives
9.2 Control Memory
9.3 Hardwired Control and Micro Programmed Control Unit
9.3.1 Microprogrammed Control
9.4 Address Sequencing
9.4.1 Conditional Branching
9.4.2 Instruction Mapping
9.4.3 Subroutines
9.5 Microprogram Sequencing
9.5.1 Micro Instruction Format
9.5.2 Symbolic Micro Instructions
9.6 Summary
9.7 Key Terms
9.8 Check Your Progress

9.0 Introduction

A control unit drives the corresponding processing hardware by generating a set of signals that are in sync with the master clock. The two major operations performed by the control unit are instruction interpretation and instruction sequencing.

Control unit is a part of the Central Processing Unit (CPU). The CPU is divided into arithmetic logic unit and the Control unit. The control unit generates the appropriate timing and control signals to all the operations involved with a computer. The flow of data between the processor, memory, and other peripherals are controlled using timing signals of the control unit.

The main function of a control unit is to fetch the data from the main memory, determine the devices and the operations involved with it, and produce control signals to execute the operations The functions of control unit are as follows:

- It helps the computer system in the process of carrying out the stored program instructions.
- It interacts with both the main memory and arithmetic logic unit.
- It performs arithmetic or logical operations.
- It coordinates with all the activities related to the other units and the peripherals.

As discussed earlier, the processor contains a number of registers and special function registers for temporary storage purposes, in addition to the arithmetic logic unit and control unit. Program Counters (PC), Instruction Registers (IR), Memory Address Registers (MAR) and Memory Data Register (MDR) are special function registers. Figure 9.1 depicts these special function registers. PC is one of the main registers in the CPU. The instructions in a program must be executed in the right order to obtain the correct results. The sequence of instructions to be executed is maintained by the PC.

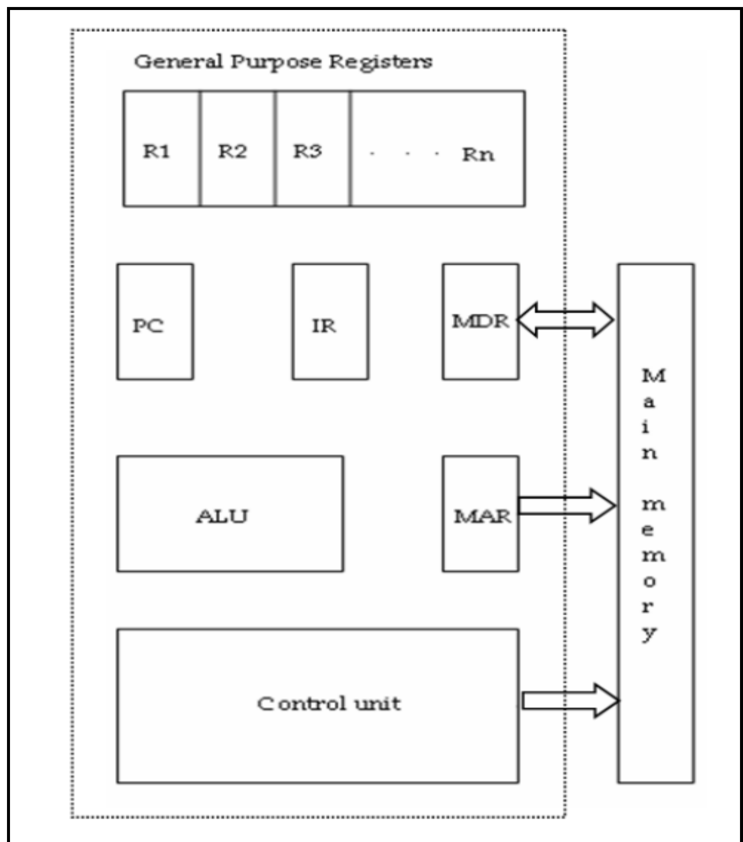


Figure 9.1 Special Function Registers of the CPU

The IR holds the instruction that is presently being executed. The timing signals generated by the control unit are based on the content of IR. The signals help in controlling the various processing elements that are necessary to execute the instruction.

The function of the other registers MAR and MDR is to transfer data. The address of the main memory to/from which data is transferred is stored in MAR. The data that is to be read/written from the specified address to the main memory is stored in MDR.

9.1 Unit Objectives

After studying this unit, you will be able to:

- Explain control memory
- Discuss hardwired control and micro programmed control unit
- Describe address sequencing
- Elaborate on microprogram sequencing

9.2 Control Memory

A control memory is a part of the control unit. Any computer that involves micro programmed control consists of two memories. They are the main memory and the control memory. Programs are usually stored in the main memory by the users. Whenever the programs change, the data is also modified in the main memory. They consist of machine instructions and data.

On the other hand, the control memory consists of micro programs that are fixed and cannot be modified frequently. They contain micro instructions which specify the internal control signals required to execute register micro operations. The machine instructions generate a chain of micro instructions in control memory. Their function is to generate micro operations that can fetch instructions from main memory, compute the effective address, execute the operation, and return control to fetch phase and continue the cycle. Figure 9.2

represents the general configuration of a microprogrammed control organization.

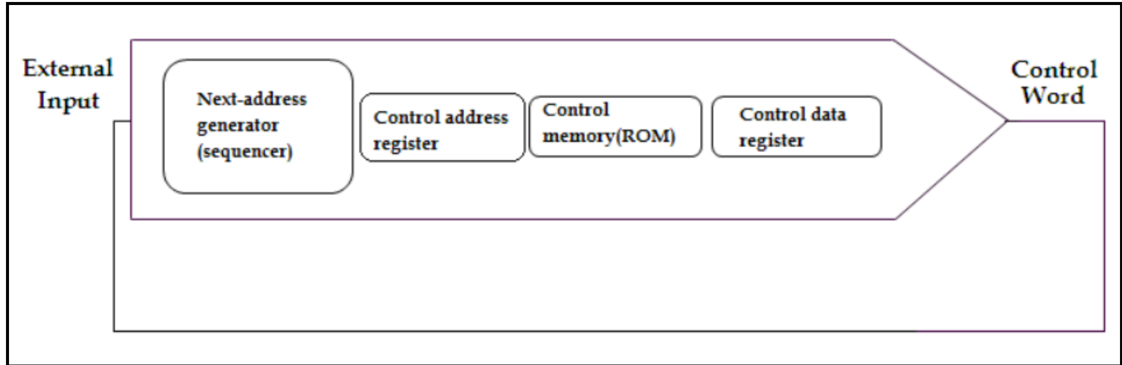


Figure 9.2 Microprogrammed Control Organization

Here, the control is presumed to be a Read Only Memory (ROM), where all the control information is stored permanently. ROM provides the address of the micro instruction. The other register, that is, the control data register stores the micro instruction that is read from the memory. It consists of a control word that holds one or more micro operations for the data processor. The next address must be computed once this operation is completed. It is computed in the next address generator. Then, it is sent to the control address register to be read. The next address generator is also known as the ***micro program sequencer***. Based on the inputs to a sequencer, it determines the address of the next micro instruction. The micro instructions can be specified in a number of ways.

The main functions of a micro program sequencer are as follows:

1. Increment the control register by one.
2. Load the address from control memory to control address register.
3. Transfer external address or load an initial address to begin the start operation.

The data register is also known as ***pipeline register***. It allows two operations to be performed at the time. It allows performing the micro operation specified

by the control word and also the generation of the next micro instruction. A dual phase clock is required to be applied to the address register and the data register. It is possible to apply a single phase clock to the address register and work without the control data register.

The main advantage of using a micro programmed control is that, if the hardware configuration is established once, no further changes can be done. However, if a different control sequence is to be implemented, a new set of micro instructions for the system must be developed.

9.3 Hardwired Control and Micro Programmed Control Unit

A hardwired control is a mechanism of producing control signals using ***Finite State Machines (FSM)*** appropriately. It is designed as a sequential logic circuit. The final circuit is constructed by physically connecting the components such as gates, flip flops, and drums. Hence, it is named as a ***hardwired controller***.

Figure 9.3 depicts a 2-bit sequence counter, which is used to develop control signals. The output obtained from these signals is decoded to generate the required signals in a sequential order.

The hardwired control consists of a combinational circuit that outputs desired controls for decoding and encoding functions. The instruction that is loaded in the IR is decoded by the ***instruction decoder***. If the IR is an 8 bit register, then the instruction decoder generates 2^8 (256) lines. Inputs to the encoder are given from the instruction step decoder, external inputs, and condition codes. All these inputs are used and individual control signals are generated. The end signal is generated after all the instructions get executed. Furthermore, it results in the resetting of the control step counter, making it ready to generate the control step for the next instruction.

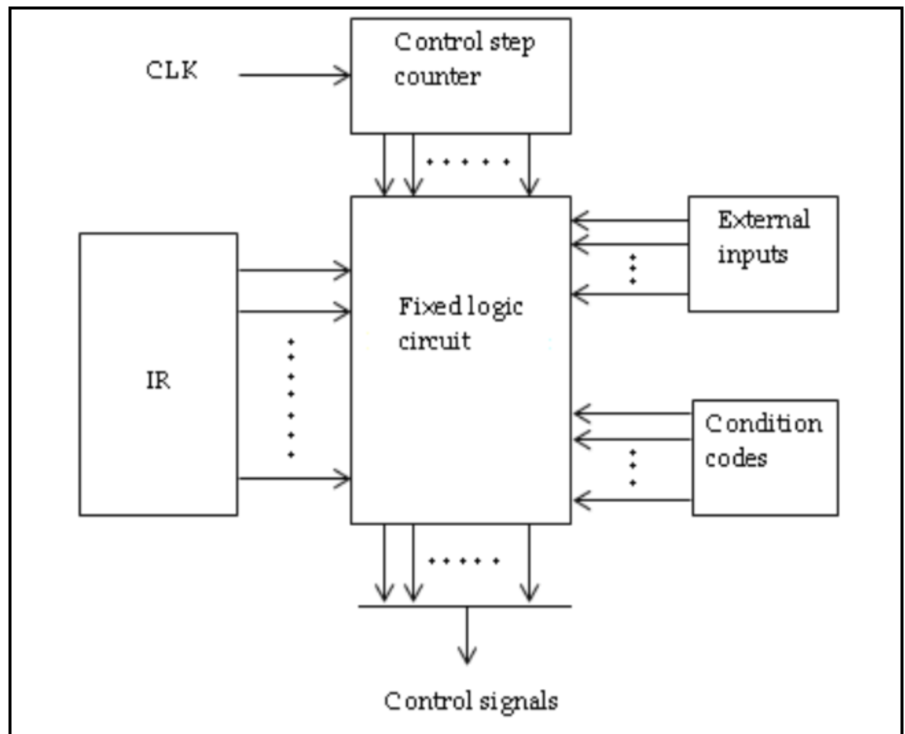


Figure 9.3 Sequential Counter

The major goal of implementing the hardwired control is to minimize the cost of the circuit and to achieve greater efficiency in the operation speed. Some of the methods that have come up for designing the hardwired control logic are as follows:

1. ***Sequence Counter Method:*** This is the most convenient method employed to design the controller of moderate complexity.
2. ***Delay Element Method:*** This method is dependent on the use of clocked delay elements for generating sequence of control signals.
3. ***State Table Method:*** This method involves the traditional algorithmic approach to design the controller using classical state table method.

When the complexity of the control function increases, it is difficult to debug the hardwired controller.

9.3.1 Microprogrammed Control

A control unit whose binary control values are stored as words in memory is known as a ***microprogrammed control unit.***

A controller results in the instructions to be executed by generating a specific set of signals at each system clock beat. Each of these output signals causes one micro operation such as register transfer. Here, the sets of control signals are said to cause specific micro operations that can be stored in the memory. Each bit that forms the micro instruction connects to one control signal. When the bit is set, the control signal is active. When it is cleared the control signal becomes inactive. These micro instructions in a sequence can be stored in the internal 'control' memory. Basically, the control unit of a micro program controlled computer is a computer within a computer.

The steps followed by the micro programmed control are:

1. To execute any instruction, the CPU must break it down into a set of sequential operations (each stating a register transfer level (RTL). These sets of operations are known as micro instruction. The sequential micro operations use the control signals to execute. (These are stored in the ROM).
2. Control signals stored in the ROM are accessed to implement the instructions on the data path. These control signals are used to control the micro operations concerned with a micro instruction that is to be executed at any time step.
3. The address of the micro instruction that is to be executed next is generated.
4. The previous 2 steps are repeated until all the micro instructions related to the instruction in the set are executed.

The address that is provided to the control ROM originates from the micro counter register. The micro counter gets its inputs from a multiplexer that selects the output of an address ROM, a current address incrementer, and address that is stored in the next-address field of current micro instruction.

Advantages of Microprogrammed Control

- More systematic design of the control unit.
- Easier to debug and modify.
- Retains the underlying structure of the control function.
- Makes the design of the control unit much simpler. Therefore, it is cheaper and less error prone.
- Orderly and systematic design process.
- Control function implemented in software and not hardware.
- More flexible.
- Complex functions are carried out easily.

Disadvantages of Microprogrammed Control

- Flexibility is achieved at extra cost.
- It is slower than a hardwired control unit.

In a micro programmed control, the control memory is assumed to be ROM, where all the data is stored permanently. The memory address of the control unit denotes the address of micro instruction.

The micro instruction has a control word. The ***control word*** denotes the operations for the data processor. After the completion of these operations the next address must be determined by the control. The next address may be the one that is next in sequence or the one that is located elsewhere. Due to this reason, it is required that some bits of the present microinstruction are used in the next instruction. Another term for the next address generator is micro program sequencer. The present address is held by the control data register until the next address is computed and read from the memory. The data register is also called pipeline register. A two phase clock is required for the same.

Table 9.1 Difference between Hardwired Control and Micro Programmed Control

Hardwired Control	Microprogrammed Control
It is not possible to modify the architecture and instruction set, once it is built.	It is possible to make modifications by changing the microprogram stored in the control memory.
Designing a computer is complex.	Designing a computer is simplified.
Architecture and instruction set is not specified.	Architecture and instruction set is specified.
It is faster.	It is comparatively slower.
It has a processor to generate signals to be implemented in correct sequence.	It uses the micro sequencer from which instruction bits are decoded and implemented.
It works through the use of drums, flip flops, flip chips, and sequential circuit.	It controls the sub devices such as ALU, registers, buses, instruction registers.

9.4 Address Sequencing

Micro instructions are stored in control memory in groups. These groups define routines. Each computer instruction has its own micro program routine that is used to generate micro operations. These micro operations are used to execute instructions. The hardware involved controls the address sequencing of the micro instructions of the same routine. They also branch the micro instructions.

Following are the steps that the control undergoes while executing a computer instruction:

1. When power is turned on, an address is initially loaded into the control address register. (This is the address of the first micro instruction).

2. The control address register is incremented resulting in sequencing the fetch routine.
3. After the fetch routine, the instruction is present in the IR of the computer.
4. Next, the control memory retrieves the effective address of the operand from the routine. (The effective address computation routine is achieved through branch micro instruction. It depends on the status of mode bits of instruction. After its completion, the address is made available in the address register).
5. Then, the mapping process happens from the instruction bits to a control memory address.
6. Based on the opcodes of instruction the micro instructions of the processor registers are generated. Each of these micro instructions has a separate micro program routine stored. The instruction code bits are transformed into the address where the routine is located and is called the ***mapping process***. A mapping procedure converts the micro instruction into a control memory address.
7. Next, subroutines are called and procedures are returned.
8. After the completion of the routine, the control address register is incremented to sequence the instruction that is to be executed. They also depend on values of status bits in processor registers. External registers are required by micro programs to store the return address that uses subroutines. After the instruction is executed, the control returns to the fetch routine. This is done by branching the micro instruction to the first address in the fetch routine.

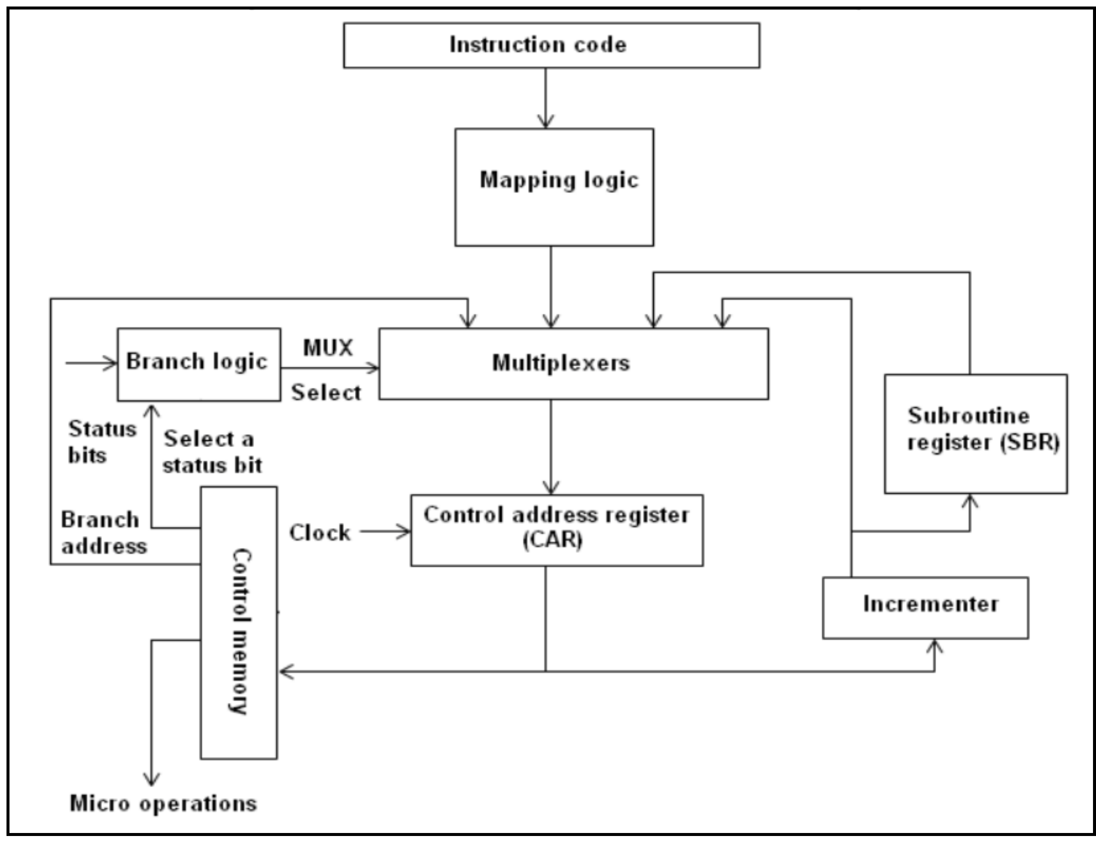


Figure 9.4 Selection of Address for Control Memory

Figure 9.4 depicts the block diagram of a control memory and its related hardware to help in selecting the next micro instruction. The micro instruction present in the control memory has a set of bits that help to initiate the micro operations in registers. They also have bits and the method that can be used to obtain the instruction of the next address. Four different paths are displayed in the figure from where the control address register retrieves its address. The CAR is incremented by the incrementer and selects the next instruction. In one of the fields of the micro instruction, the branching address can be specified to result in branching. To determine the condition of the status bits of micro instruction, conditional branching may be used. A mapping logic circuit is used to transfer an external address. A special register is used to store the return address, so that whenever the micro program wants to return from subroutine, it can use the value from the special register.

9.4.1 Conditional Branching

From the figure 9.4, the flow of control is clear. The carry out of an adder, mode bits of an instruction, Input/Output status conditions are the special bits of status conditions. Based on conditions whether their value is 0 or 1, their information is tested and actions are initiated. These bits combine with the other fields of the micro instructions and control the decisions regarding the conditions in the branch logic.

The hardware associated with branch logic can be employed in a number of ways. One of the common ways is to test the condition, and if it is satisfied, then branch to the specified address. Else, increment the address register. A multiplexer can be employed to work through the branch logic.

Suppose the number of status bit conditions = 8. Out of the eight bits, three are used to specify selection variables for the multiplexer. If selected status bit = 1, multiplexer output = 1, else it would be 0. When the MUX O/P = 1, it produces a control signal and transfers the branch address to CAR from micro instruction. When MOX O/P = 0, the address register gets incremented.

Unconditional Branch Micro Instruction

The unconditional branch micro instruction can be achieved by transferring the branch address from control memory to CAR. Here, the status bit at the input of MUX is fixed to 1. When there is a reference to these status bit lines, the branch address is loaded into CAR causing the unconditional branching.

9.4.2 Instruction Mapping

When a micro instruction specifies the branch to the first word in the control memory where the routine for micro instruction is placed, it leads to a special type of branch. The branch has its status bits placed in the instruction's opcode part.

As depicted in figure 9.5 the instruction format of a simple computer has an opcode of four bits. They can specify up to 16 different instructions, if the control memory has 128 words that require a 7-bit address. Each of the operations has a micro program routine that helps in executing the instruction.

A mapping process can transform a 4-bit opcode to a 7-bit address for control memory. In this process, a 0 is placed in the most significant bit of the address, 4 opcode bits are transferred and 2 least significant bits are cleared. This way each computer instruction has a micro program routine that has a capacity to group 4 micro instructions.

Sometimes, a mapping function needs to use the integrated circuit called ***Programmable Logic Device (PLD)***. It uses the AND/OR gates that consist of electrical fuses internally. They are commonly implemented in the mapping function that is expressed in terms of Boolean expressions.

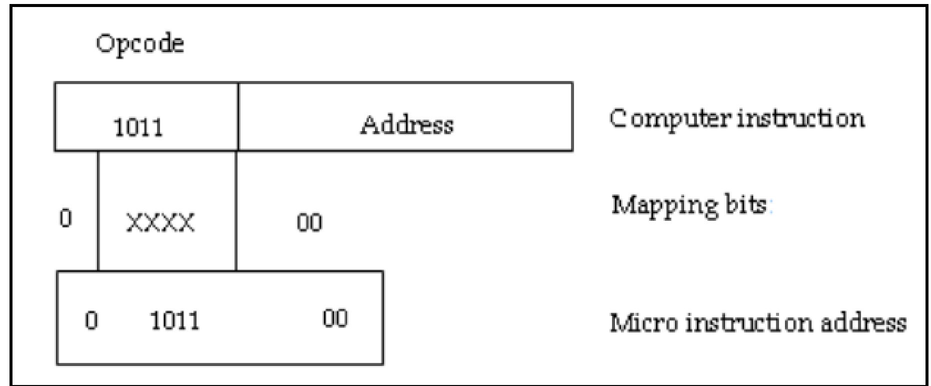


Figure 9.5 Instruction Code to Microinstruction Address Mapping

9.4.3 Subroutines

Certain tasks cannot be performed by the program alone. They need additional routines known as subroutines. Subroutines are routines that can be called within the main body of the program at any point of time. There are cases when many programs contain identical code sections. These code sections can be saved in subroutines and used wherever common codes are used. For example, the code needed to generate an effective address of operand for a sequence of microinstruction is common for all the memory reference instructions. This code can be a subroutine and called from within other routines.

All the micro programs that implement subroutine must have extra memory space to store the return address. The extra space is called the subroutine

register. It is used to store the return address during a subroutine call and restore during subroutine return. The incremented output must be placed in a subroutine register from a CAR. This must be branched to the beginning of subroutine. Now, the register becomes a means of transferring the address to return to the main routine. The registers must be arranged in the LIFO (Last In First Out) stack so that it is easy to get the addresses.

9.5 Microprogram Sequencing

The micro code for the control memory must be generated by the designer once the configuration of a computer is established. The generation of code is called ***micro programming***.

The important points to be considered while designing the micro program sequencer are: ***Size of micro instruction*** and ***Time of address generation***.

The micro instruction's size must be in the least, so that the control memory required is less and the cost is reduced. Micro instructions can be executed at a faster rate if the time to generate an address is less. This results in increased throughput.

The disadvantages of microprogram sequencing are as follows:

- If each machine instruction has a separate micro routine, then it results in the usage of larger areas for storage.
- The branching requires more time for execution.

Consider an instruction Add X, AR. This instruction will require four addressing modes - register, auto increment, auto decrement, and indexing in case of indirect forms.

Computer Configuration

The micro code for the control memory is generated after the computer configuration and its microprogrammed control unit is established. The figure 9.6 displays the simple digital computer and the way it is micro programmed. There are two memory units, the instructions and data is stored in the main

memory and the micro programs are stored in the control memory. The processor unit consists of four registers, Program Counter (PC), Address Register (AR), Data Register (DR), and an Accumulator (AC). The control unit contains two registers. They are a Control Address Register (CAR) and a Subroutine Register (SBR).

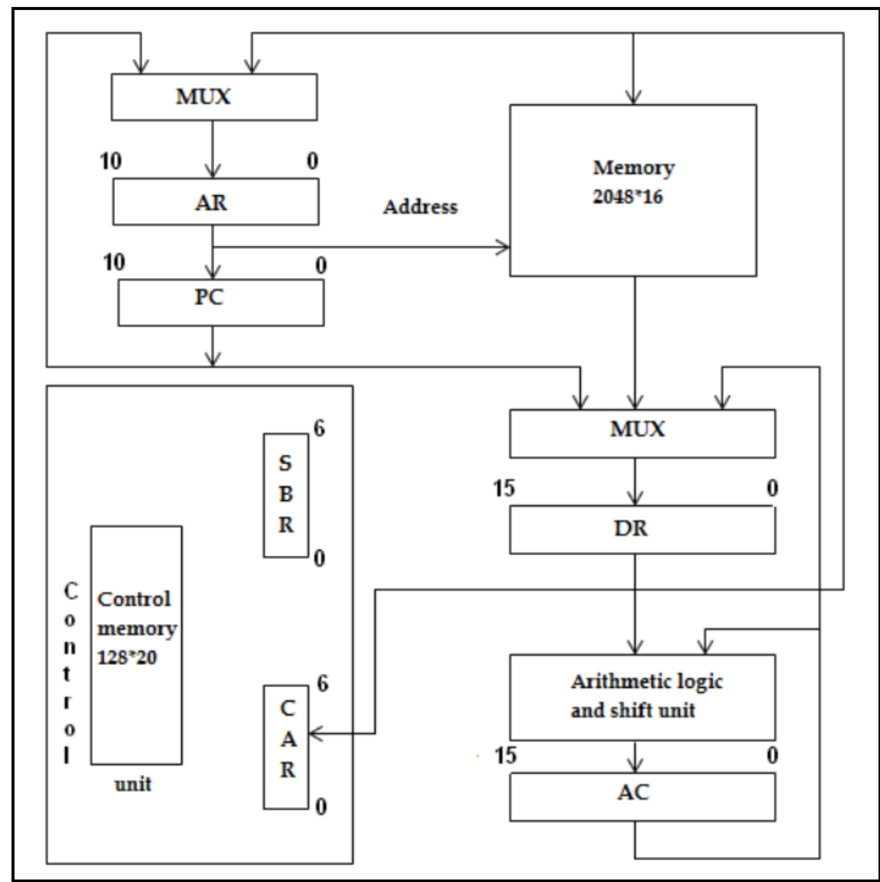


Figure 9.6 Configuration of Computer Hardware

As depicted in figure 9.6, multiplexers are used to transfer information within the registers in the processor. **AR** can get data from **PC** or **DR**. **DR** can receive data from **AC**, **PC**, or **memory**. **The PC** gets data only from the **PC**. The data from **AC** and **DR** can undergo arithmetic and logic operations and be placed in the **AC**. The **DR** is the source of data for memory, where the data that is read can go to **DR** and no other register.

9.5.1 Micro Instruction Format

A micro instruction format consists of 20 bits in total. They are divided into four parts as shown in the figure 9.7. F1, F2, F3 are the micro operation fields. They specify micro operations for the computer. CD is the condition for branching. They select the status bit conditions. BR is the branch field. It specifies the type of branch. AD is the address field. It contains the address field whose length is 7 bits. The micro operations are further divided into three fields of three bits each. These three bits can specify seven different micro operations. Each micro instruction can have only three micro operations, one from every field. If it uses less than three, it will result in more than one operation using the no operation binary code.

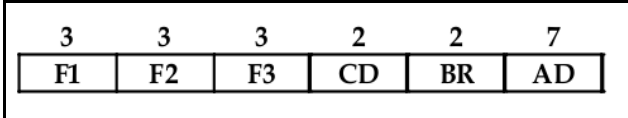


Figure 9.7 Micro Instruction Code Format

Consider **F2** and **F3** as two consecutive micro operations specified by microinstructions. It does not describe **F1**.

$$DR \leftarrow M\,[AR] \text{ with } F2 = 100$$

$$PC \leftarrow PC + 1 \text{ with } F3 = 101$$

Here, the micro operation will be **000 100 101**. The **F1** field remains **000** as nothing is specified for the same. Also, two or more conflicting microoperations cannot be specified consecutively.

Condition Field

A condition field consists of 2 bits. They are encoded to specify four status bit conditions. As specified in the table, the first condition is always a **1**, with **CD = 0**. The symbol used to denote this condition is **'U'**. The table 6.3 depicts the different condition fields and their descriptions in a clear manner.

Table 9.2 Condition Field Symbols and Descriptions

	Condition	Symbol	Comments
00	Always =1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC=0	Z	Zero value in AC

As depicted in table 9.2, when the condition 00 is combined with **BR** (branch) field, it results in unconditional branch operation. After the execution is read from memory the **indirect bit I** is available from **bit 15** of **DR**. The status of the next bit is provided by the **AC** sign bit. If all the bits in **AC** are **1**, then it is denoted as **Z** (its binary variable whose value is 1). The symbols **U**, **I**, **S** and **Z** are used to denote status bits while writing microprograms.

Branch Field

The BR (branch) field consists of 2 bits. It is used by combining with the AD (address) field. The reason for combining with the AD field is to choose the address for the next micro instruction. The table 9.3 explains the different branch fields and their functions.

Table 9.3 Branch Field Symbols and Descriptions

BR	Symbol	Function
00	JMP	CAR ? AD, if condition = 1
		CAR ? CAR + 1, if condition = 0
01	CALL	CAR ? AD, SBR ? CAR + 1, if condition = 1
		CAR ? CAR + 1, if condition =0
10	RET	CAR ? SBR (return from subroutine)
11	MAP	CAR (2 - 5) ? DR (11 - 4), CAR (0, 1, 6) ? 0

As depicted in table 9.3, when BR = 00, a JMP operation is performed and when BR = 01, a subroutine is called. The only difference between the two instructions is that when the micro instruction is stored, the return address is stored in the Subroutine Register (SBR). These two operations are dependent

on the CD field values. When the status bit condition of the CD field is specified as 1, the address that is next in order is transferred to CAR. Else, it gets incremented. If the instruction wants to return from the subroutine, its BR field is specified as 10. This results in the transfer of the return address from SBR to CAR. The opcode bits of instruction can be mapped with an address for CAR if BR field is 11. They are present in DR (11 - 14) after an instruction is read from memory. The last two conditions in the BR fields are not dependent on the CD and AD field values.

9.5.2 Symbolic Micro Instructions

The micro instructions can be specified by symbols. It is translated to its binary format with an assembler. The symbols must be defined for each field in the micro instruction. Furthermore, the users must be allowed to define their own symbolic addresses. Every line in an assembly language defines a symbolic instruction. These instructions are split in five fields, namely, label, microoperations, CD, BR, and AD as explained in table 9.4.

Table 9.4 Fields and their Descriptions

Field	Description
Label	It may be empty or specified by a symbolic address. It is ended by a colon (:).
Micro operations	It consists of one or more symbols separated by commas. But each F field consists of only a single symbol.
CD	It has one of the letters U, I ,A, or Z.
BR	It consists of one among the four symbols defined in table 6.4
AD	It specifies the value for address field of micro instruction in any of the following ways described below: - With address in symbols that appears as label. - With NEXT symbol that point to the next address in a sequence. - With BR field containing a Ret or a MAP symbol, the AD field is left empty and converted to seven zeroes by the assembler.

Each micro instruction implements the internal register transfer operation shown by the register transfer representation. The representation in symbols is necessary while writing micro programs in assembly language format. The actual internal content which is stored in the control memory is in binary representation. In normal practice, programs are written in symbolic form initially and later converted into binary using an assembler.

9.6 Summary

- A control unit controls the data flow through the processor and coordinates the activities of the other units that are involved with it.
- The processor consists of many registers within it. One of the main registers is Program Counter (PC). It holds the instruction that is to be executed next in a sequence. The function of the other registers such as MAR and MDR is to transfer data.
- Control memory is a part of the control unit. It stores all the micro programs that cannot be modified frequently. They are fixed programs.
- The data register is also known as pipeline register. It allows two operations to be performed at a time. It allows performing the micro operation specified by the control word and also the generation of the next micro instruction.
- The hardwired control uses the finite state machines to generate control signals.
- Micro programmed control is another way of generating control signals. They consist of a sequence of micro instructions that correspond to a sequence of steps in an instruction execution. The next address generator is called a micro program sequencer.
- Each computer instruction has its own micro program routine that is used to generate microoperations. An address sequencer is a circuit used to generate addresses for accessing the memory device.
- Subroutines are the additional routines that are used by programs to perform some tasks.

9.7 Key Terms

- ***Micro Instruction:*** An elementary instruction that controls the sequencing of instruction execution and the flow of data in a processor. Execution of a machine language instruction requires the execution of a series of micro instructions.
- ***Micro Program:*** It consists of a sequence of micro instructions corresponding to the sequence of steps in the execution of a given machine instruction.
- ***Micro Programming:*** It is the method of generating control signals by setting the individual bits in a control word of a step.

9.8 Check Your Progress

Q1) What is Microprogram sequencing? Also list the disadvantages of it.

Q2) Define Address sequencing. How are subroutines beneficial in address sequencing?

Q3) Discuss the advantages and disadvantages of microprogrammed control.

Q4) Differentiate between Hardwired Control and Microprogrammed Control Unit.

Q5) Discuss the concept of Instruction Mapping.

References

Computer Organization and Architecture, Radhakrishnan, T., & Rajaraman, V. (2007), RajKamal Electric Press, New Delhi.

Digital Electronics, 3rd ed., Godse A.P & Godse D.A. (2008), Technical Publications, Pune.

MODULE: V

INPUT/ OUTPUT ORGANIZATION, PARALLEL PROCESSING AND MULTIPROCESSORS

Unit 10 – Input/Output Organization

Structure

10.0 Introduction
10.1 Unit Objectives
10.2 Basic Input/ Output Structure of Computers
10.3 Synchronous and Asynchronous Data Transfer
10.3.1 Strobe Control
10.3.2 Handshaking
10.4 Serial and Parallel Communication
10.5 Modes of Transfer
10.5.1 Programmed I/O (Polling)
10.5.2 Interrupt Driven I/O
10.5.3 Direct Memory Access (DMA)
10.6 Priority Interrupt
10.6.1 Daisy- Chain Priority
10.6.2 Parallel Priority Interrupt
10.6.3 Priority Encoder
10.7 Device Drivers
10.8 Standard I/O Interfaces (Buses)
10.9 Bus Arbitration
10.10 I/O Processor
10.11 Summary
10.12 Key Terms
10.13 Check Your Progress

10.0 Introduction

The most essential building blocks of any computer are the processor, memory system, and input/output modules. There is a need for some peripheral devices for the computer to establish communication with the other devices and the external world. In the desire to obtain a basic minimum working system, there must be some mechanism to input command or data by the user, and this is achieved through input devices attached through input ports of the computer system. One of the most widely and frequently used input devices for any

computer is its keyboard. Similarly, to get results or some feedback from the computer, one or more devices, known as output devices, are interfaced through the output port of the system. As an example, the monitor or the LCD screen of any computer, where we generally look for computer-generated results or computer-generated feedback.
However, apart from keyboard and display, there are other types of devices, which are interfaced with a computer. Many times they need some special considerations for their interfacing and one of such important considerations is the maximum allowable speed of data transfer popularity known as bandwidth. For example, a video interface needs data input at a very high speed to maintain the quality of the on-screen animation at an acceptable standard. The same demand is applicable for any hard disc drive also. On the other hand, a keyboard may be taken as a very slow interface for data transactions. In this chapter, we discuss various techniques to interface all the widely used different input and output devices.

10.1 Unit Objectives

On completion of this unit, the reader will be able to:

- Study the different techniques of I/O communication.
- Know about the Polling, Interrupt driven, and DMA methods of data transfer.
- Understand the details of standard communication buses and other I/O interfaces.
- Learn the basics of bus arbitration and I/O processor.

10.2 Basic Input/output Structure of Computers

According to any processor, any device other than a memory device is an input/output (I/O) device. For example, devices like timer, interrupt controller, USART, DMA controller are found within the motherboard of any computer. However, from the user's point of view, I/O devices are something like a keyboard, printer, mouse, CRT, and so on. We will now discuss both types of

peripheral units and study the method of communication necessary for the smooth operation of the computer.

Interfacing and Communication Techniques:

All peripheral devices have their own processors within the devices. Therefore, communication between a host (computer) and any one of its peripheral units is essentially the communication between two processors. This communication link is always established through the wire-connections. This means that the processors are not placed within the same circuit and-may only be interconnected through some cable or multiple wires. The general structure of such an interface is shown in Figure 10.1. It can be observed that the processors in the motherboard of the host (computer) and all other processors within peripherals are generally different and have their own operating frequencies. Secondly, these processors are never directly interconnected but through some I/O ports. All necessary signals and data between any two processors are transferred through these ports. The method of data communication between the host and its peripheral may be any one of the following three:

- Programmed I/O
- Interrupt driven I/O
- Direct Memory Access (DMA)

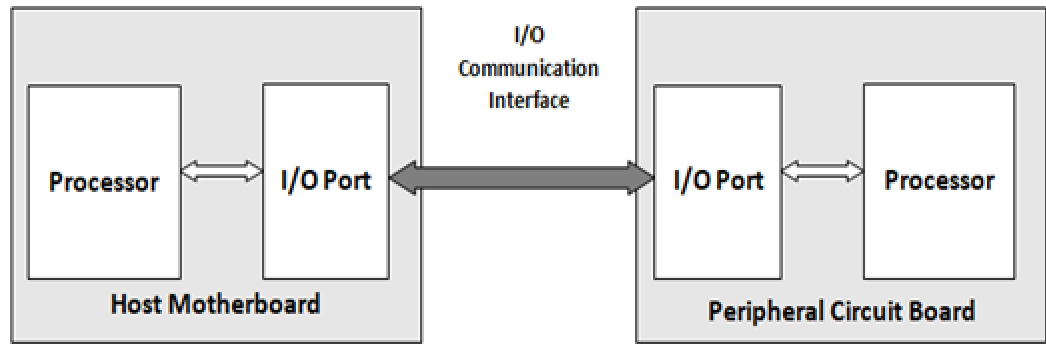


Figure 10.1 Representation of the link between a Host and its peripheral unit

The quantity of data transacted by the first two cases is comparatively lesser than that by the DMA. In the *programmed I/O*, the processor knows when to transmit, but generally has no idea when to receive the data from the external source and, therefore, sometimes the processor waits for data reception. In *interrupt-driven I/O*, the transaction begins with an interrupt from the peripheral device, which receives or transmits data from or to the host. Both of these types of transactions are controlled by the processor of the host. In the *DMA controller*, the processor of the host temporarily ends the system bus to the DMA controller, which takes care of the mass data transaction between a peripheral device and the memory of the host.

Classification of Communication:

On the basis of different characteristics, classification of data communication adopted in computers is represented in Figure 10.2. The technique of data transmission in programmed I/O and interrupt-driven I/O is generally asynchronous, and during DMA, it is synchronous. Moreover, for a few peripheral devices, the information communication might need to be in serial format, whereas for alternative cases it would be in parallel format.

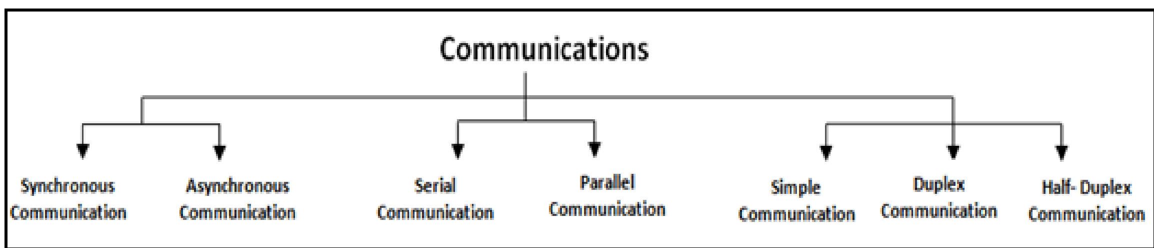


Figure 10.2 Classification of Data Communication

In simplex communication, the direction of data transfer is always unidirectional, while in full-duplex communication it is always bidirectional. In half-duplex communication, the data transmission is bidirectional but time-shared, i.e., unidirectional at any time. We will now discuss some features of asynchronous communications and serial and parallel communications.

10.3 Synchronous and Asynchronous Data Communication

In ***synchronous communication***, a standard clock governs the communication between any two devices. The simplest example is the communication between the processor and its main memory where both must obey the system clock and complete the data transaction as per the predefined schedule.

In ***asynchronous communication***, two devices at two ends of data transmission have their own individual clocks. Due to the absence of any common clock source between these two, there cannot be any predefined time duration for completion of data transfer from one end to another. Therefore, in general, the transmission is completed with the help of some additional handshaking signals.

For example, if devices A and B are to communicate in an asynchronous manner (assuming data flow from A to B), then initially A asks B (through some signal) “Are you ready”? Device A now keeps on waiting till it gets the reply from B like “I am ready”. Once it is assumed that B is ready to accept, A sends data properly along with another signal, which may be interpreted as “Here is the data. Please accept it”. After receiving the data device B finally sends the acknowledgement signal to A, as if saying, “I have received the data thank you”. After receiving the acknowledgement, the signal from B, A starts sending another set of data. Note that in this communication, the start of each phase is dependent upon the proper completion of the previous phase. A will not send another fresh set of data until it receives the acknowledgment signal from B. Therefore, in asynchronous communication, extra handshaking signals are necessary to complete the process without any fault, in spite of two devices operating under different clock frequencies.

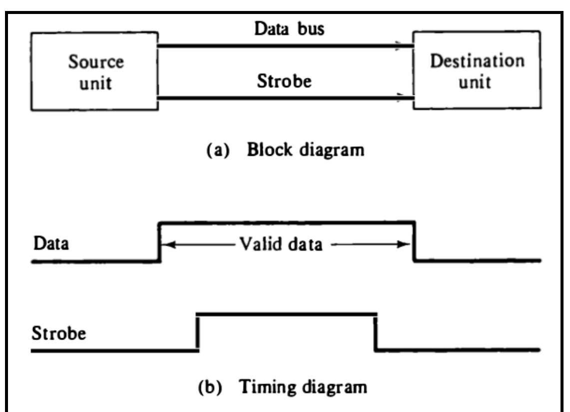
Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indi-cate the time at which data is being transmitted. One way of achieving this is by means of a ***strobe*** pulse supplied by one of the units to indicate to the other unit when the transfer has to occur. Another method commonly used is to

accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This type of agreement between two independent units is referred to as ***handshaking***.

The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers. In fact, they are used extensively on numerous occasions requiring the transfer of data between two indepen-dent units. In the general case we consider the transmitting unit as the source and the receiving unit as the destination. For example, the CPU is the source unit during an output or a write transfer and it is the destination unit during an input or a read transfer. It is customary to specify the asynchronous transfer between two independent units by means of a timing diagram that shows the timing relationship that must exist between the control signals and the data in the buses. The sequence of control during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination unit.

10.3.1 Strobe Control

The strobe control method of asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated by either the source or the destination unit. The data bus carries the binary information from the source unit to the destination unit. Typically, the bus has multiple lines to transfer an entire byte or word. The strobe is a single line that informs the destination unit when a valid data word is available in the bus. In many computers the strobe pulse is actually controlled by the clock pulses in the CPU. The CPU is always in control of the buses and informs the external units how to transfer data.



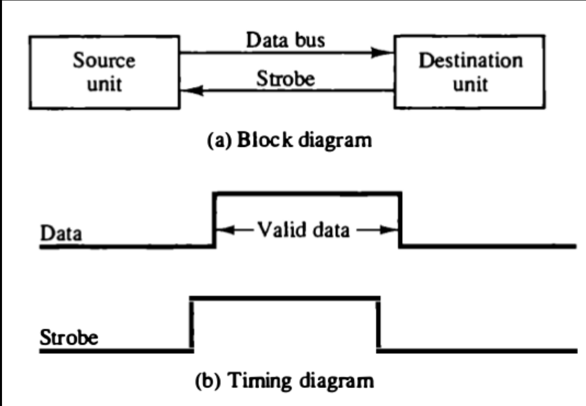


Figure 10.3 (i) Source-initiated (ii) Destination-initiated strobe for data transfer

10.3.2 Handshaking

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that initiates the transfer. The basic principle of the ***two-wire handshaking*** method of data transfer is as follows. One control line is in the same direction as the data flow in the bus from the source to the destination. It is used by the source unit to inform the destination unit whether there is valid data in the bus. The other control line is in the other direction from the destination to the source. It is used by the destination unit to inform the source whether it can accept data. The sequence of control during the transfer depends on the unit that initiates the transfer.

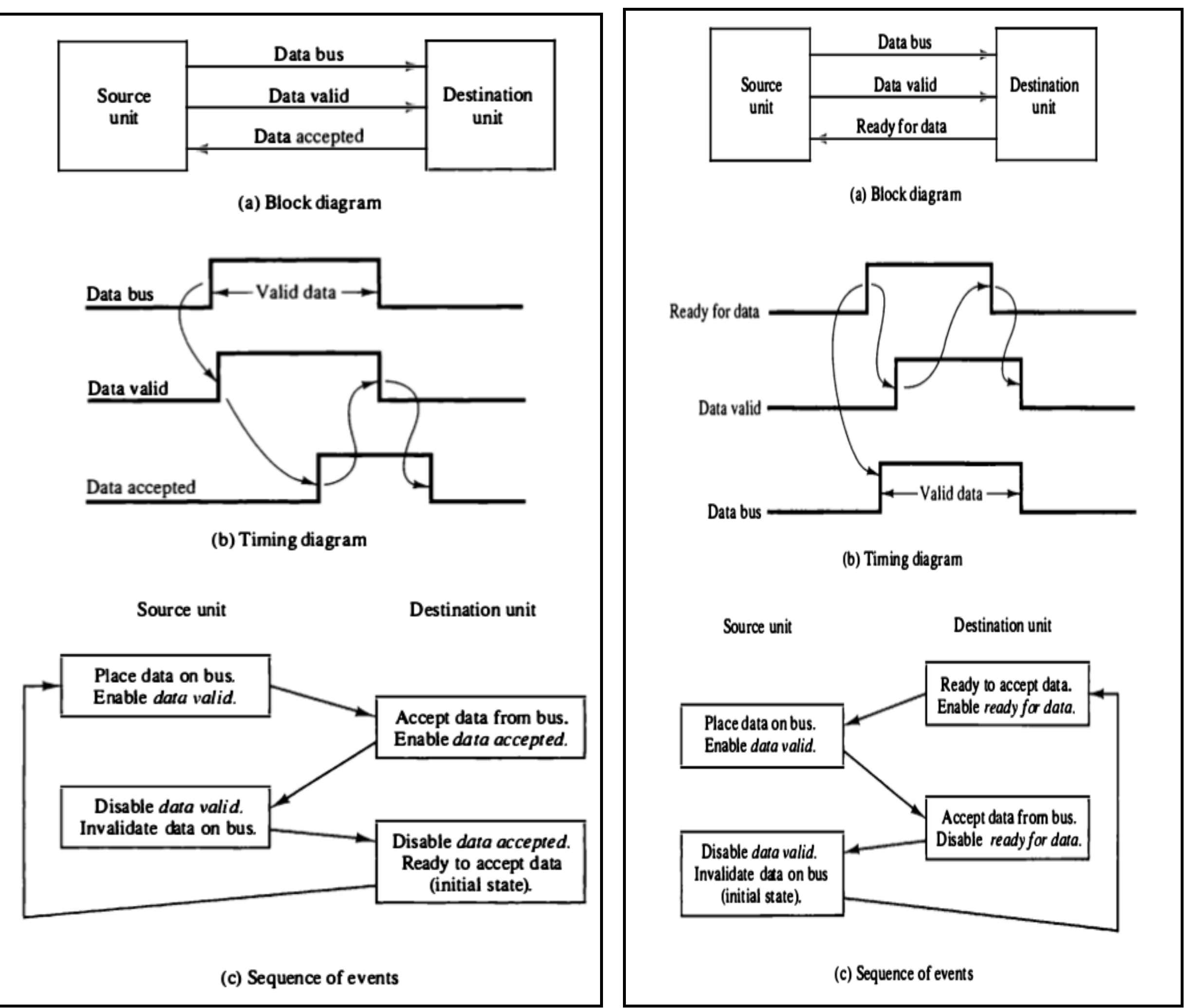


Figure 10.4 (i) Source-initiated (ii) Destination-initiated transfer using handshaking

Figure 10.4 (i) shows the data transfer procedure when initiated by the source. The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the destination unit. The timing diagram shows the exchange of signals between the two units. The sequence of events listed in part (c) shows the four possible states that the system can be at any given time. The source unit initiates the transfer by placing the data on the bus and enabling its data valid signal. The data accepted signal is activated by the destination unit after it accepts the data from the bus. The source unit then disables its data valid signal, which invalidates the data on the bus. The

destination unit then disables its data accepted signal and the system goes into its initial state. The source does not send the next data item until after the destination unit shows its readiness to accept new data by disabling its data accepted signal. This scheme allows arbitrary delays from one state to the next and permits each unit to respond at its own data transfer rate. The rate of transfer is determined by the slowest unit.

The destination-initiated transfer using handshaking lines is shown in figure 10.4 (ii). Note that the name of the signal generated by the destination unit has been changed to ready for data to reflect its new meaning. The source unit in this case does not place data on the bus until after it receives the ready for data signal from the destination unit. From there on, the handshaking proce-dure follows the same pattern as in the source-initiated case. Note that the sequence of events in both cases would be identical if we consider the ready for data signal as the complement of data accepted. In fact, the only difference between the source-initiated and the destination-initiated transfer is in their choice of initial state.

The handshaking scheme provides a high degree of flexibility and reliabil-ity because the successful completion of a data transfer relies on active partic-ipation by both units. If one unit is faulty, the data transfer will not be timeout completed. Such an error can be detected by means of a timeout mechanism, which produces an alarm if the data transfer is not completed within a prede-termined time.

10.4 Serial and Parallel Communications

Asynchronous communication could also be more classified as serial and parallel, relying upon a variety of bits being transferred at the identical instant. In serial communication, only one bit of information is transferred at a time, while in parallel communication, multiple bits, generally 8 or 12 or 16, are transferred at the same time. Therefore, the requirement of the number of transmission lines necessary to connect two communicating devices would be less for serial communication than for parallel communication. However, the

speed of data transfer would be faster in the case of parallel communication than the speed of serial communication.

Format of Serial Data Transfer: In the case of serial communication, as the information to be transmitted or received one bit at a time byte of information is broken and sent bit-by-bit, as shown in Figure 10.5, with the least significant bit (bit 0) transmitted first and the most significant bit (bit 7) transmitted last. At the receiving end, transmitted bits are sampled in the middle of their transmission. Note that one start bit and one or two stop bits are added with every byte of transmitted data. This start bit has always the same logic level as that of logic 0, i.e., the start bit is always low. On the contrary, the logic of stop bit(s) would always be high (1). Therefore, if one stop bit is used, to send one byte of information, 10 bits instead of 8 bits must be transmitted. In serial communication, the rate of data transmission, known as the *baud rate*, has some standard values. These values are 300, 600, 1200, 2400, 4800, 9600, and 19200 bits per second (bps).

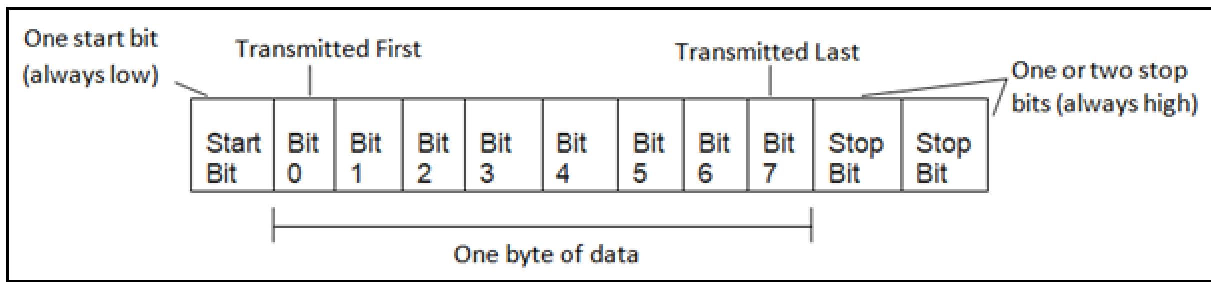


Figure 10.5 Format of Serial Data Transfer

These start and stop bits are added with data bytes before the transmission and deleted from the received data to regain the transmitted byte after data reception. As the distance to be covered by transmission is larger in the case of serial communication used in computers, the signals are generally buffered and biased to eliminate noise and other transient disturbances.

USART: The universal synchronous asynchronous receiver transmitter (USART) is used to make serial communication easier. It is time-consuming for any processor at the transmitting end to load every byte to be transmitted (from

memory), break it into several bits, add start bit and stop bit(s) and finally transmit all these, one bit at a time. The same is also valid if the processor is at the receiving end. In that case, it has to discard the start bit and stop bit(s), assemble each byte, bit by bit, and then store it at a certain location of memory or use it for some immediate purpose. To cultivate this issue, USART is used.

The USART contains one transmitting terminal (known as TxD) and another receiving terminal (known as RxD) at its external end. These terminals must be interfaced with similar and appropriate terminals of another USART at the other end of the communication. The internal end of USART has necessary terminals so that it may be interfaced with its processor through address, data, and control signals of the CPU bus. Internally, the USART has at least two buffers, one for input and the other for output. The processor writes data byte, which needs to be transmitted out through serial interface, within this output buffer. Similarly, the receiving buffer of the USART keeps on collecting the serially received data, bit by bit, and when it has collected all eight bits, the processor reads this input buffer. When the receiving buffer is full, the complete byte is copied to the second buffer and the receiving buffer keeps on collecting fresh incoming bits, offering the CPU more time to read the just-stored byte of information.

10.5 Modes of Transfer

Binary information received from an external device is usually stored in mem-ory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit. Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; others transfer the data directly to and from the memory unit. Data transfer to and from peripherals may be handled in one of three possible modes:

1. Programmed I/O

2. Interrupt-initiated I/O
3. Direct memory access (DMA)

10.5.1 Programmed I/O (Polling)

Programmed I/O is the simplest technique of transferring data from CPU to the computer peripherals. Polling is another term used for the programmed I/O method of data transfer. The exchange of data is carried out through an I/O module, a program that gives direct control to the I/O operations such as sending a write or read command, observing the device status, transferring the data, etc.

The complete operation of the programmed I/O can be summarized as:

- While a program execution, the processor runs into instruction for I/O operation.
- An appropriate command is issued to the I/O module by the processor to execute that instruction.
- After performing the requested action of the I/O module, the processor will set the suitable bits in the I/O status register.
- This status will be checked periodically by the processor until the operation is complete.

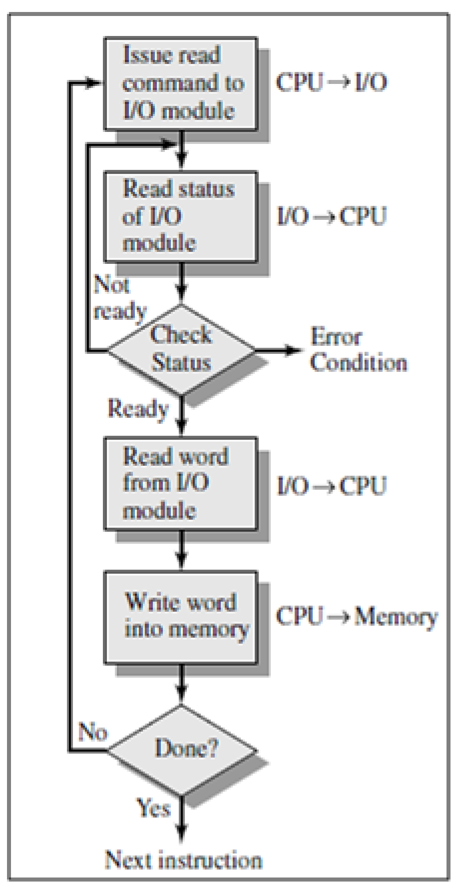


Figure 10.6 Flowchart for Programmed I/O Method for data input operation

(**Source-** Computer Organization and Architecture, Eighth Edition, William Stallings, Chapter-7, Page no. 226)

The process of sending the data out through the programmed I/O method is simpler than to receive it. While sending the data, the processor simply writes the relevant data in the latch of the output port. The data is immediately available at the outer end of the port latch to be shared with the external world. While receiving the data through programmed I/O, the situation changes, as it is expected that the data is being generated from some outside source which is uncontrollable by the processor. The processor simply waits to receive the data through the port. This waiting consumes time and keeps the processor busy needlessly. This situation can be avoided by using another method called Interrupt driven I/O.

10.5.2 Interrupt Driven I/O

As already discussed, the disadvantage of a programmed I/O method is that it keeps the processor busy or waits for a long time to get the concern of the I/O module for reception or transmission of data. This can degrade the overall performance of the system. To overcome this issue, Interrupt driven I/O technique is used. Interrupt driven I/O is a method of controlling the I/O activity until the peripheral device sends a signal to receive or send the data. Whenever it is decided that the device is ready to transfer data, it gives an Interrupt request signal to the system. On receiving this external interrupt signal, the processor branches to the service program to execute the I/O data transfer, while halting the task it is already performing. Once it is done, it again resumes to the previous task. The method gives more efficiency but requires a complex hardware and software system.

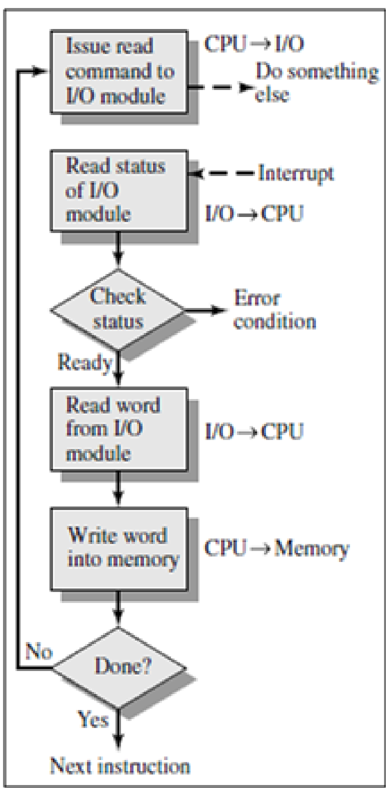


Figure 10.7 Flowchart for Interrupt driven I/O Method for data input operation

(**Source-** Computer Organization and Architecture, Eighth Edition, William Stallings, Chapter-7, Page no. 226)

Figure 10.7 represents a flowchart for interrupt processing in a computer system. The process is as follows:

- A peripheral device initiates a request to the I/O module, while the processor is performing some other task.
- The processor now receives the interrupt signal on the interrupt request service line.
- After receiving the request, the processor halts the present task and reads the status of the I/O module and checks the error condition.
- After processing an interrupt request, the processor returns to its prior task.

The advantage of using interrupt-driven I/O data transfer method is that the processor need not spend its valuable time to watch an input for a long time. Rather it may keep itself busy in other important tasks and the external interrupt from the concerned device simply draws its attention and completes the data transaction process as and when required. However, the problem may occur when the number of interrupting peripheral devices is more than the number of available external interrupt-input pins of the processor. There may be multiple I/O modules so it becomes difficult for the processor to determine the device issuing the interrupt request and is unable to prioritize which interrupt request has to be approached first. In such situations, an external interrupt controlling device needs to be interfaced with the processor so that more number of interrupting peripheral devices may share the same interrupt input of the processor. Intel 8259 is one such device, which may be interfaced with most Intel processors to enhance the interrupt handling capability.

10.5.3 Direct Memory Access (DMA)

Though the Interrupt-driven I/O method of data transmission is more efficient than the programmed I/O method, it still involves the processor to transmit data between memory and the I/O module. The following factors affect the efficiency of a processor in both these techniques:

- The speed of the I/O module is dependent on the speed of the processor.

- The processor is engaged in executing a number of instructions for one I/O transfer request and affects the data processing and other tasks.

Direct Memory Access (DMA) is the third type of technique used for transferring data without approaching the processor. This method establishes data transmission within the main memory and external device and does not involve the processor in this operation. DMA is more efficient to move a large volume of data. It improves the processor activity and the data transfer rate by not involving the processor and establishing a direct link to the memory and the external devices. DMA can be implemented by setting an additional module on the system bus. This DMA module can be used instead of using the processor. As the data to be transferred to and from the memory using a system bus, the DMA module can use this system bus only when the processor is not using it or can force the processor to halt its operation for some time. This forcing the processor technique is called ***cycle stealing*** and is used most commonly.

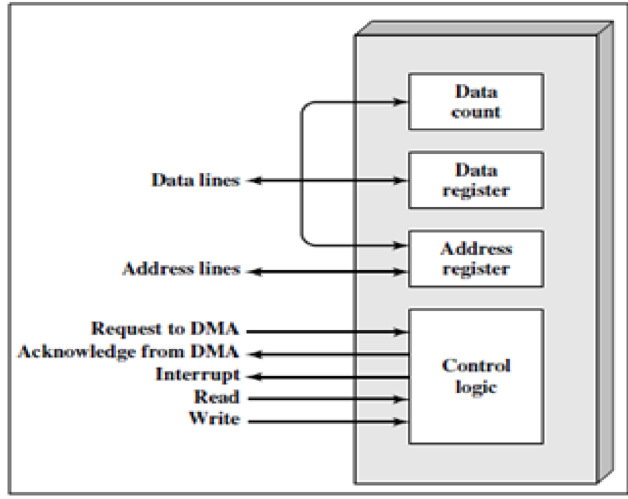


Figure 10.8 Block diagram of DMA

(**Source-** Computer Organization and Architecture, Eighth Edition, William Stallings, Chapter-7, Page no. 237)

The processor sends a command to the DMA module, whenever it has to read or write a set of data. It includes:

- A read or write command sent through read and write control lines between the processor and the DMA module.

- The number of words that are to be read or written through data lines and are stored in the data counter register.
- The starting location is communicated through data lines and is stored in the address register. This starting location corresponds to the memory to read from and write to.
- Data lines communicating the address of the corresponding I/O device.

After sending this information, the processor continues with other work. The DMA module now transfers the entire set of data directly to/ from memory without approaching the processor. After completing the data transfer, the DMA module issues an interrupt signal to the processor informing about the completion of the task using the same system bus. It should be noted that the involvement of the processor is only at the beginning and end of the data transfer.

The above mechanism of the DMA module can be configured in different ways, such as:

1. **Single Bus, detached DMA:** In this mechanism, all the modules share a single system bus and the DMA module uses programmed I/O method of data transfer between the memory and the I/O module. Due to programmed I/O, in this method, each word transfer consumes two bus cycles, so this method is less efficient.
2. **Single Bus, integrated DMA:** This configuration corresponds to a path between the DMA module and the I/O modules that are exclusive of the system bus. The number of bus cycles can be decreased by keeping the DMA module as a part of an I/O module or a separate module controlling one or more I/O modules. The data transfer between the DMA and I/O modules is done beyond the system bus.
3. **I/O Bus:** This configuration is improved from the other two configurations. In this method, the I/O modules and the DMA modules are connected through an I/O bus. This helps in reducing the number of I/O interfaces in the DMA module. This configuration is easily

expandable. Like an integrated DMA method, the data transfer takes place off the system bus.

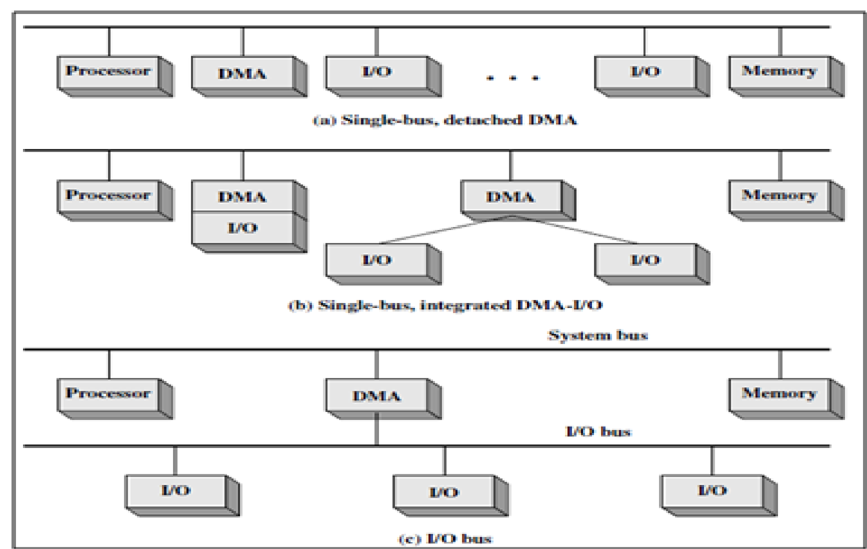


Figure 10.9 Different DMA Configurations

(**Source-** Computer Organization and Architecture, Eighth Edition, William Stallings, Chapter-7, Page no. 238)

10.6 Priority Interrupt

Data transfer between the CPU and an I/O device is initiated by the CPU. However, the CPU cannot start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal. The CPU responds to the interrupt request by storing the return address from the PC into a memory stack and then the program branches to a service routine that processes the required transfer. Some processors also push the current ***PSW (program status word)*** onto the stack and load a new PSW for the service routine. We neglect the PSW here in order not to complicate the discussion of VO interrupts.

In a typical application a number of I/O devices are attached to the computer, with each device being able to originate an interrupt request. The first task of the interrupt system is to identify the source of the interrupt. There is also the possibility that several sources will request service simultaneously. In this case the system must also decide which device to service first.

A ***priority interrupt*** is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously. The system may also determine which condi-tions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to requests which, if delayed or interrupted, could have serious consequences. Devices with high-speed transfers such as magnetic disks are given high priority, and slow devices such as keyboards receive low priority. When two devices interrupt the computer at the same time, the computer services the device, with the higher priority first.

Establishing the priority of simultaneous interrupts can be done by software or hardware. A ***polling*** procedure is used to identify the highest-priority source by software means. In this method there is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt. The highest-priority source is tested first, and if its interrupt signal is on, control branches to a service routine for this source. Otherwise, the next-lower-priority source is tested, and so on.

A hardware priority-interrupt unit functions as an overall manager in an interrupt system environment. It accepts interrupt requests from many sources, determines which of the incoming requests has the highest priority, and issues an interrupt request to the computer based on this determination. To speed up the operation, each interrupt source has its own interrupt vector to access its own service routine directly. Thus no polling is required because all the decisions are established by the hardware priority-interrupt unit. The hardware priority function can be established by either a serial or a parallel connection of interrupt lines. The serial connection is also known as the daisy--chaining method.

10.6.1 Daisy-Chain Priority

The daisy-chaining method of establishing priority consists of a serial connec-tion of all devices that request an interrupt. The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain. This method of connection between three devices and the CPU is shown in figure 9.10. The interrupt request line is common to all devices and forms a wired logic connec-tion. If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU. When no interrupts are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU. This is equivalent to a negative-logic OR operation.

The CPU responds to an interrupt request by enabling the interrupt acknowledge line . This signal is received by device 1 at its ***PI (priority in)*** input. The acknowledge signal passes on to the next device through the ***PO (priority out)*** output only if device 1 is not requesting an interrupt. If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output. It then proceeds to insert its own interrupt ***vector address (VAD)*** into the data bus for the CPU to use during the interrupt cycle.

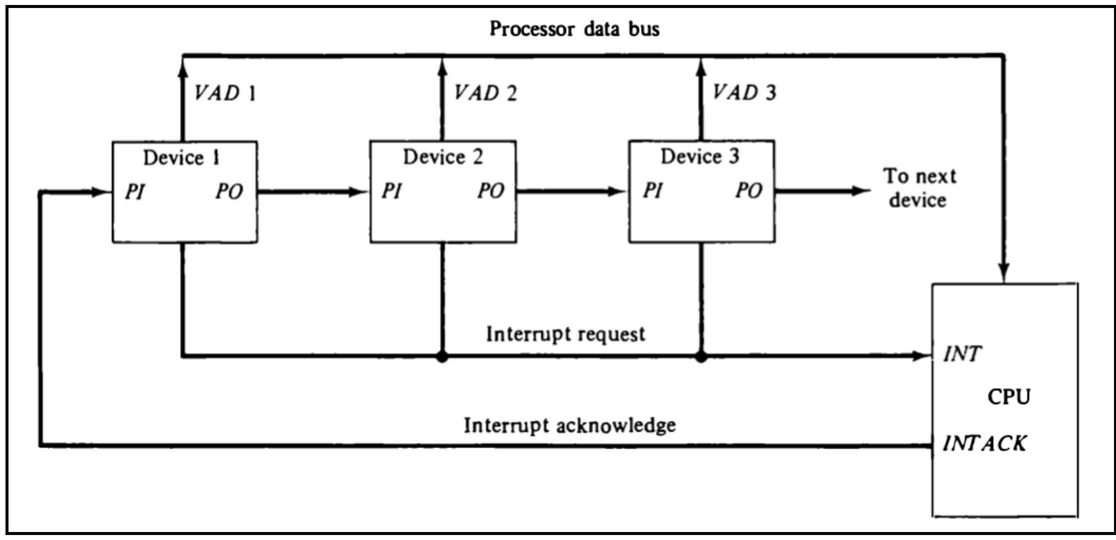


Figure 10.10 Daisy- chain priority interrupt

10.6.2 Parallel Priority Interrupt

The parallel priority interrupt method uses a register whose bits are set sepa-rately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. In addition to the interrupt register, the circuit may include a mask register whose purpose is to control the status of each interrupt request. The mask register can be programmed to disable lower-priority interrupts while a higher-priority device is being serviced. It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower-priority device is being serviced.

The priority logic for a system of four interrupt sources is shown in figure 10.11. It consists of an interrupt register whose individual bits are set by external conditions and cleared by program instructions. The magnetic disk, being a high-speed device, is given the highest priority. The printer has the next priority, followed by a character reader and a keyboard. The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register. Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way an interrupt is recognized only if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU.

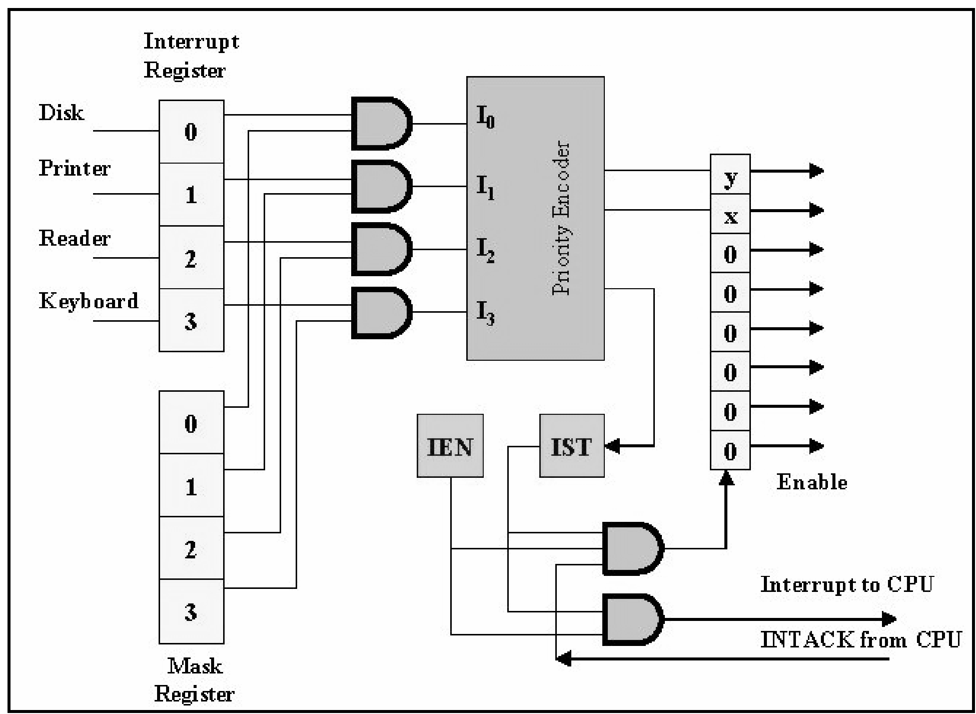


Figure 10.11 Parallel Priority Interrupt

10.6.3 Priority Encoder

The priority encoder is a circuit that implements the priority function. The logic of the priority encoder is such that if two or more inputs arrive at the same time, the input having the highest priority will take precedence. The truth table of a fow--input priority encoder is given in Table 10.1. The x's in the table designate don't-are conditions. Input 10 has the highest priority; so regardless of the values of other inputs, when this input is 1, the output generates an output ry 00. I_1 has the next priority level. The output is 01 if $I_1 = 1$ provided that $I_0 = 0$, regardless of the values of the other two lower-priority inputs. The output for 12 is generated only if higher-priority inputs are 0, and so on down the priority level. The interrupt status IST is set only when one or more inputs are equal to 1. If all inputs are 0, IST is cleared to 0 and the other outputs of the encoder are not used, so they are marked with don't-care conditions. This is because the vector address is not transferred to the CPU when IST = 0. The Boolean functions listed in the table specify the internal logic of the encoder.

Usually, a computer will have more than four interrupt sources. A priority encoder with eight inputs, for example, will generate an output of three bits. The output of the priority encoder is used to form part of the vector address for each interrupt source. The other bits of the vector address can be assigned any value. For example, the vector address can be formed by appending six zeros to the x and y outputs of the encoder. With this choice the interrupt vectors for the four I/O devices are assigned binary numbers 0, 1, 2, and 3.

Table 10.1 Priority Encoder Truth Table

Inputs				Outputs			Boolean functions
I_0	I_1	I_2	I_3	x	y	IST	
1	×	×	×	0	0	1	
0	1	×	×	0	1	1	$x = I'_0 I'_1$
0	0	1	×	1	0	1	$y = I'_0 I_1 + I'_0 I'_2$
0	0	0	1	1	1	1	$(IST) = I_0 + I_1 + I_2 + I_3$
0	0	0	0	×	×	0	

10.7 Device Drivers

A device driver is a software or rather a group of subroutines, related to a particular device and used by the operating system as and when required. For example, let us take the case of a keyboard. Whenever the computer is switched on, there must be a subroutine to initialize the keyboard too. Moreover, whenever the keys of the keyboard are pressed, an interrupt is generated, which must be serviced by the processor of the computer. Subroutines related to these operations of reset initialization and interrupt service are stored in a suitable place, generally within the hard disc, as per the direction and knowledge of the operating system. The operating system labels these subroutines as the device driver for the peripheral devices and evokes the necessary one as per the system demand.

Therefore, we may readily conclude that every device must have its own device driver. Moreover, the same device with a different manufacturer or version should have a different driver. For example, the device driver for a Microsoft mouse may not be suitable for a Logitech mouse. That is the reason for supplying the related driver with any peripheral device, which must be properly stored by the operating system at the time of its installation.

10.8 Standard I/O Interfaces (Buses)

A computer system uses different interface standards for different operations. The transmission of data from one place to another within or outside the processor is carried out using different buses. The processor bus is defined on the chip itself. The peripheral devices like main memory are in need of high-speed and direct connection with the processor. But, there is a limitation to connect all the devices to the processor directly. For this purpose, another bus is used by the processor to support more devices. These two buses are interconnected with each other by a circuit, known as a ***bridge***. Different companies have designed this external bus which is used with the main processor bus for the efficient operation of the system. The Peripheral Component Interconnect (PCI) bus was one of the first manufactured I/O interface buses. The other I/O interfaces are Universal Serial Bus (USB), Small Computer System Interface (SCSI) bus, General Purpose Interface Bus (GPIB), or Hewlett Packed Interface Bus (HPIB), Versa Module Euro card (VME) bus, Multi-bus, etc. Each one was developed by different manufacturers, and have different advantages and disadvantages. Here, we are going to discuss only three main I/O interface buses.

Peripheral Component Interconnect (PCI) bus

The PCI bus is independent of the main processor bus but provides the same function to the devices connected to it as the processor bus does. The devices connected to the PCI bus are assigned addresses in the memory address of the processor. PCI was a low-cost developed bus in the year 1992 for supporting

high-speed graphic and video devices and even microprocessor systems. The main advantage of the PCI bus is its plug-and-play feature that ensures that only the device is needed to be plugged into the PCI bus board and rest operation is carried out by the software.

Small Computer System Interface (SCSI) bus

SCSI is a standard bus introduced by the American National Standards Institute (ANSI) that works on the parallel communication technique for data transfer. A narrow SCSI bus has eight lines and transmits one-byte data at a time while a wide SCSI bus has sixteen lines and can transmit 16 bits data at a time. The devices connected to the SCSI bus do not occupy any address space in the memory of the processor like devices connected to the processor bus. A SCSI controller is used to connect the SCSI bus to the processor bus that uses DMA for data transfer from memory to the device, or vice versa. One of the key features for high performance of SCSI is handling multiple data transfer requests by using two types of controllers, initiator, and target controller. The operation of a SCSI bus is as follows:

1. The SCSI controller acts as an initiator and maintains control of the bus.
2. After the arbitration process of the initiator, the target controller is being selected to take over the control of the bus.
3. An output operation by the target starts and meanwhile the initiator sends a command for the read operation.
4. The target halts the connection between the target and the initiator temporarily and releases the bus.
5. After completing the reading operation, the target again requests for the control of the bus and restores the halted connection.
6. After transferring the data to the initiator controller, it stores the data in the memory using the DMA method. Now the SCSI controller sends an interrupt to the processor to give the message that the requested task has been completed.

Universal Serial Bus (USB)

USB is the most commonly used interface bus that also works on the feature of plug-and-play. It is capable of connecting almost all types of peripheral devices to the system. It was first introduced in 1995. USB is a four-pin interface, with one pin as a power supply terminal (+5V), a ground terminal, and the other two pins as data signals. It can interface a maximum of 127 devices. USB adopts the serial data communication technique for transferring data, unlike the SCSI bus.

10.9 Bus Arbitration

A system bus is accessed by more than one processor at a time when a multiprocessor system is considered. Therefore, this requires an appropriate mechanism to decide the priority for the processor. This is known as bus arbitration. When multiple devices place the request for communication through the same bus at the same time, the situation must be solved by arbitration as at any given time only one device can use the bus.

The situation of bus arbitration is very much similar to the situation of multiple interrupts encountered by the processor at the same time. One can recollect the case of 8085 with its five interrupt inputs. If all interrupts are acknowledged at the same time, then TRAP would be serviced first as it is assigned the highest priority. The next priority is given to the RST7.5 interrupt by default. Then it is RST6.5, RST5.5, and finally the INTR, the interrupt with the lowest priority. So, some priority is assigned for different devices attached to any bus for the purpose of the arbitration.

There are three prominent methods to resolve bus arbitration:

- Daisy chain method
- Polling method
- Independent request method

In the daisy chain method, several devices are attached to the bus and the bus-arbiter. All these devices need to share the common bus-request and the bus-

grant signal lines. It should be noted that the bus-grant signal line, unlike the bus-request signal line, is generated from the bus-arbiter and passes through the switch of each device to the next device. This method has a simple design and less number of control lines. The method can be disadvantageous when there is a propagation delay due to the serial bus arrangement. Due to this, if one device fails then the entire system may fail.

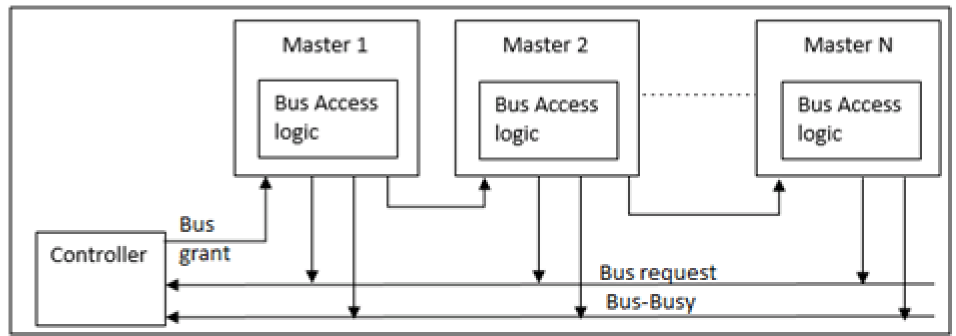


Figure 10.12 Daisy Chain Method of Bus Arbitration

(Source- https://www.ques10.com/p/11191/what-is-the-bus-arbitration-what-are-different-m-1/)

In the polling method, all the devices use the same line for requesting the bus. In response to this request, the controller polls the devices by sending a series of bus master addresses on address lines. One of the devices at a time recognizes its address and activates the bus busy lines. In this method the priority of the device is flexible and there is no dependency of the devices on each other. The only issue with this method is that adding the bus masters increases the number of address lines in the circuit.

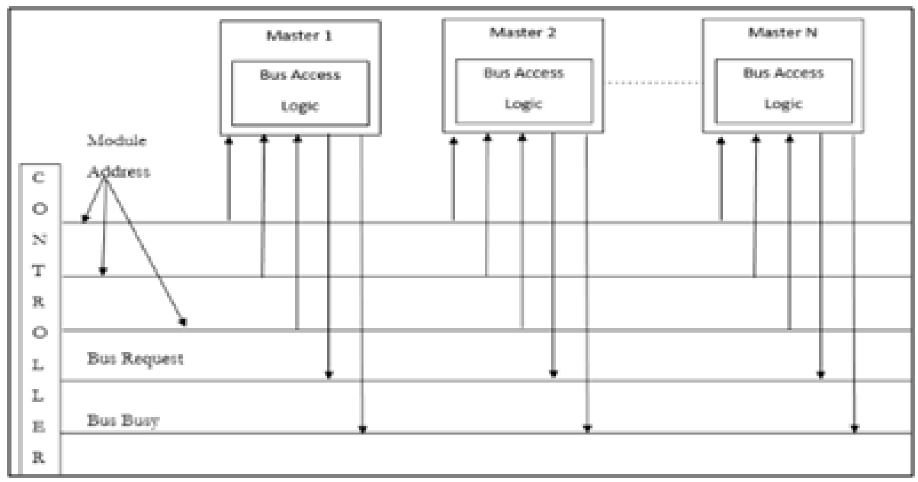


Figure 10.13 Polling Method of Bus Arbitration

(Source- https://www.ques10.com/p/11191/what-is-the-bus-arbitration-what-are-different-m-1/)

The independent request method of bus arbitration offers individual bus request and bus grant lines to each device of the system. The controller knows which device has been requested and the bus is granted to that device. The priority of the bus is simultaneous to bus requests, provided that the bus busy line is inactive. This speeds up the process of bus arbitration and the system is independent of the number of devices connected.

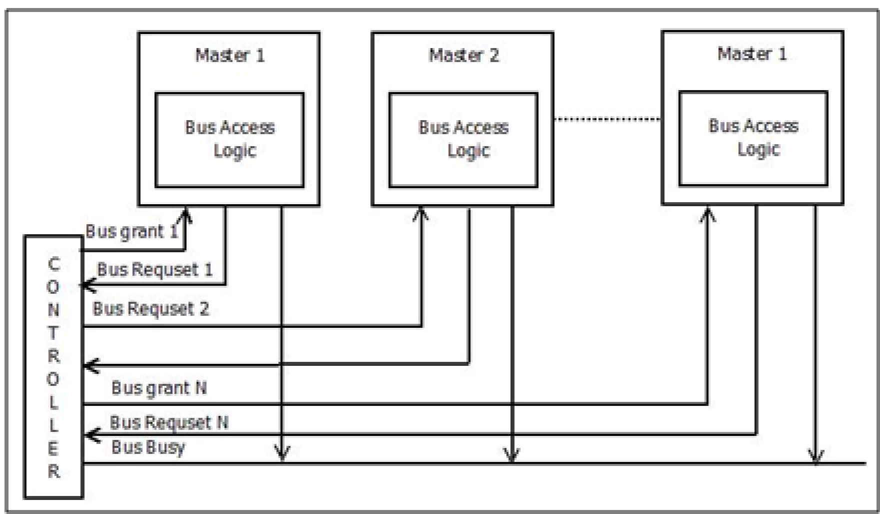


Figure 10.14 Independent Request Method of Bus Arbitration

(Source- https://www.ques10.com/p/11191/what-is-the-bus-arbitration-what-are-different-m-1/)

10.10 I/O Processor

As already discussed, the Direct Memory Access (DMA) I/O data transfer method reduces the overuse of the processor of the computer system while executing the I/O operations. DMA is the most efficient method other than the programmed I/O and Interrupt driven I/O method. DMA also provides simultaneous operation for the processor and the I/O operations that reduce the waiting time of the processor.

An Input-Output Processor (IOP) is a special-purpose processor just like a CPU that is developed to handle the I/O operations. The IOP acts as an interface between the peripheral devices and the computer system. The IOP is capable of fetching and executing instructions that are specific to I/O transfers. In addition to the I/O operations, IOP can also perform other tasks like arithmetic, logical, code translation, and branching. Figure 10.15 represents the basic block diagram of an I/O processor. It should be noted that the different processors of all peripheral devices share the main memory of the system as shown in the figure.

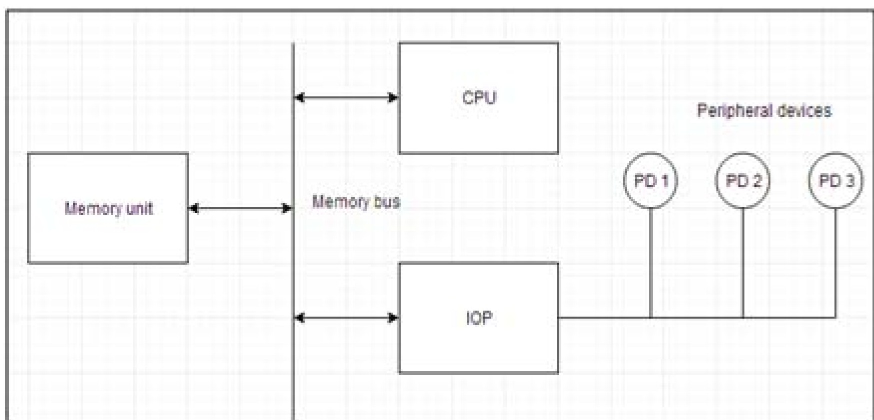


Figure 10.15 Block diagram of I/O processor

(Source- https://www.studytonight.com/computer-architecture/input-output-processor)

10.11 Summary

- On the basis of the characteristics of data, the communication can be classified as serial or parallel, synchronous or asynchronous, simplex, full-duplex, and half-duplex.
- The three major techniques of I/O data transfer are programmed I/O (Polling), Interrupt driven I/O, and DMA I/O. In the polling method, the processor is very much engaged, and the interrupt-driven method releases the extra burden from the processor. DMA is used to transfer a large volume of data and it is a separate processing unit that is least dependent on the CPU.
- All the external devices have their own device drivers, the software that is essential to initialize the device. Various standard buses are available to interface the peripheral devices with the processor, such as PCI, USB, SCSI, etc.
- Bus arbitration is necessary to deal with the multiple device requests for the same bus. There are three methods for bus arbitration, daisy chain, polling, and Independent request method.
- An Input-Output Processor (IOP) is a special-purpose processor just like CPU that is developed to handle the I/O operations. The IOP acts as an interface between the peripheral devices and the computer system.

10.12 Key Terms

- **Interrupt Controller:** An interrupt controller is a device that multiplexes multiple interrupts to one CPU line, which are accessed by setting priority for the interrupts.
- **Cycle stealing Mode:** When DMA grabs the system bus for operation from the processor, then it is said to be operating in a cycle stealing mode.
- **GPIB/ HPIB:** General Purpose Interface Bus is a type of bus interface generally used for establishing an interface between measuring

instruments and the computer system. GPIB was earlier known with the name HPIB.

- **Daisy Chain Method:** It is a type of bus arbitration method in which all the devices have to share the same bus- request and bus- grant signal lines. In this method, every device is dependent on the other device, if one device fails then the entire system fails.

10.13 Check Your Progress

Q1) Differentiate between simplex, full-duplex, and half-duplex type of communication techniques.

Q2) Write a short note on USART.

Q3) What are the advantages of DMA I/O over interrupt-driven I/O method of data transfer?

Q4) What is the difference between programmed I/O and interrupt-driven I/O?

Q5) Briefly explain the three methods of Bus arbitration.

Q6) Write a short note on:

a) DMA b) PCI Bus c) I/O processor d) SCSI Bus

Q7) Differentiate between Serial and Parallel Communication Techniques.

References

Computer Architecture and Organization, Subrata Ghoshal, Pearson Publication.

Computer Organization and Architecture, 9th edition, William Stallings, Pearson Publication.

Computer System Architecture, M. Morris Mano

https://www.geeksforgeeks.org/io-interface-interrupt-dma-mode/

https://www.ques10.com/p/11191/what-is-the-bus-arbitration-what-are-different-m-1/

http://www.idc-online.com/technical_references/pdfs/information_technology/Standard_io_interfaces.pdf

http://www.ee.ncu.edu.tw/~jfli/computer/lecture/ch05.pdf

Unit 11 – Parallel Processing

Structure

11.0 Introduction

11.1 Unit Objectives

11.2 Parallel Processing

11.3 Pipelining

11.4 Data Dependency

11.5 Handling of Branch Instructions

11.6 Vector Processing

11.7 Array Processors

11.8 Summary

11.9 Key Terms

11.10 Check Your Progress

11.0 Introduction

Traditionally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequences of instructions. Processors execute programs by executing machine instructions in a sequence and one at a time. Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation, store results). This view of the computer has never been entirely true. At the microoperation level, multiple control signals are generated at the same time. Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for a long time. Both of these are examples of performing functions in parallel.

This approach is taken further with superscalar organization, which exploits instruction-level parallelism. With a superscalar machine, there are multiple execution units within a single processor, and these may execute multiple instructions from the same program in parallel.

After an overview of parallel processing, this unit illustrates the concept of pipelining, instruction- level parallelism, vector processing and array processors. Pipeline processing is an implementation technique where arithmetic sub operations or the phases of a computer instruction cycle overlap in execution. Vector processing deals with computations involving large vectors and ma-trices. Array processors perform computations on large arrays of data.

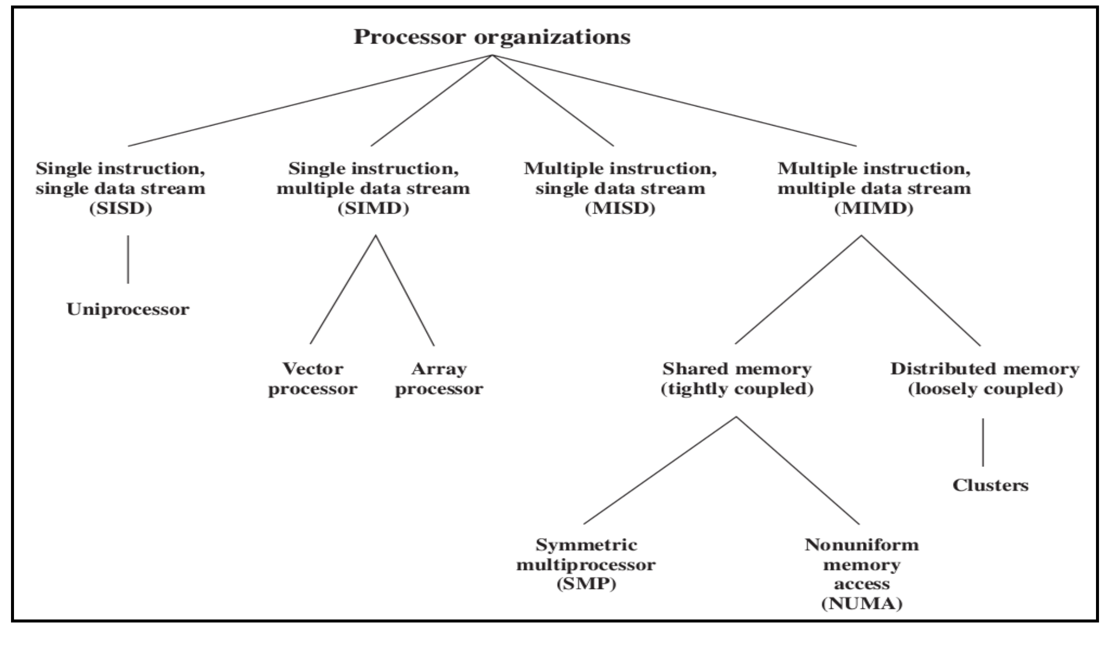
Before proceeding, let's discuss ***Flynn's classification of computers.*** Flynn's classification depends on the distinction between the perform-ance of the control unit and the data-processing unit. It emphasizes the behavioral characteristics of the computer system rather than its operational and structural interconnections. One type of parallel processing that does not fit Flynn's classification is pipelining.

A taxonomy first introduced by Flynn[1] is still the most common way of categorizing systems with parallel processing capability. Flynn proposed the following categories of computer systems:

- **Single instruction, single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory. Uniprocessors fall into this category.
- **Single instruction, multiple data (SIMD) stream:** A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data memory, so that instructions are executed on different sets of data by different processors. Vector and array processors fall into this category, and are discussed in the upcoming sections of this unit.
- **Multiple instruction, single data (MISD) stream:** A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure is not commercially implemented.
- **Multiple instruction, multiple data (MIMD) stream:** A set of processors simultaneously execute different instruction sequences on different data

[1] Flynn, M. "Some Computer Organizations and Their Effectiveness." IEEE Transactions on Computers, September 1972.

sets. SMPs (Symmetric Multiprocessors), clusters, and NUMA (Non Uniform Memory Access) systems fit into this category.



(a)

CU IS PU DS MU

(a) SISD

CU IS PU_1 DS LM_1
PU_2 DS LM_2
PU_n DS LM_n

(b) SIMD (with distributed memory)

CU_1 IS PU_1 DS
CU_2 IS PU_2 DS
CU_n IS PU_n DS
Shared memory

(c) MIMD (with shared memory)

CU = Control unit
IS = Instruction stream
PU = Processing unit
DS = Data stream
MU = Memory unit
LM = Local memory

SISD = Single instruction, single data stream
SIMD = Single instruction, multiple data stream
MIMD = Multiple instruction, multiple data stream

CU_1 IS PU_1 DS LM_1
CU_2 IS PU_2 DS LM_2
CU_n IS PU_n DS LM_n
Interconnection network

(d) MIMD (with distributed memory)

(b)

Figure 11.1 (a) A Taxonomy of Parallel Processor Architectures (b) Alternative Computer Organizations

(Source- *Computer Organization and Architecture, 9th edition,* W Stallings, Pearson Publication.**)**

11.1 Unit Objectives

On completion of this unit, the reader will be able to:

- Summarize the types of parallel processor organizations.
- Present an overview of Pipelining.
- Learn the concept of Data dependency and Instruction-level parallelism.
- Explain the concept of vector processing and array processors.

11.2 Parallel Processing

Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.

For example, while an instruction is being executed in the ALU, the next instruction can be read from memory. The system may have two or more ALUs and be able to execute two or more instructions at the same time. Furthermore, the system may have two or more processors operating concurrently.

The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time. The amount of hardware increases with parallel processing and with it, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

Parallel processing can be viewed from various levels of complexity. At the lowest level, we distinguish between parallel and serial operations by the type of registers used. Shift registers operate in serial fashion one bit at a time, while registers with parallel load operate with all the bits of the word simulta-neously. Parallel processing at a higher level of complexity can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously.

Parallel processing is established by distributing the data among the multiple functional units. For example, the arithmetic, logic, and shift operations can be separated into three units and the operands diverted to each unit under the supervision of a control unit. Figure 11.2 represents one positive way of separating the execution unit into eight functional units operating in parallel.

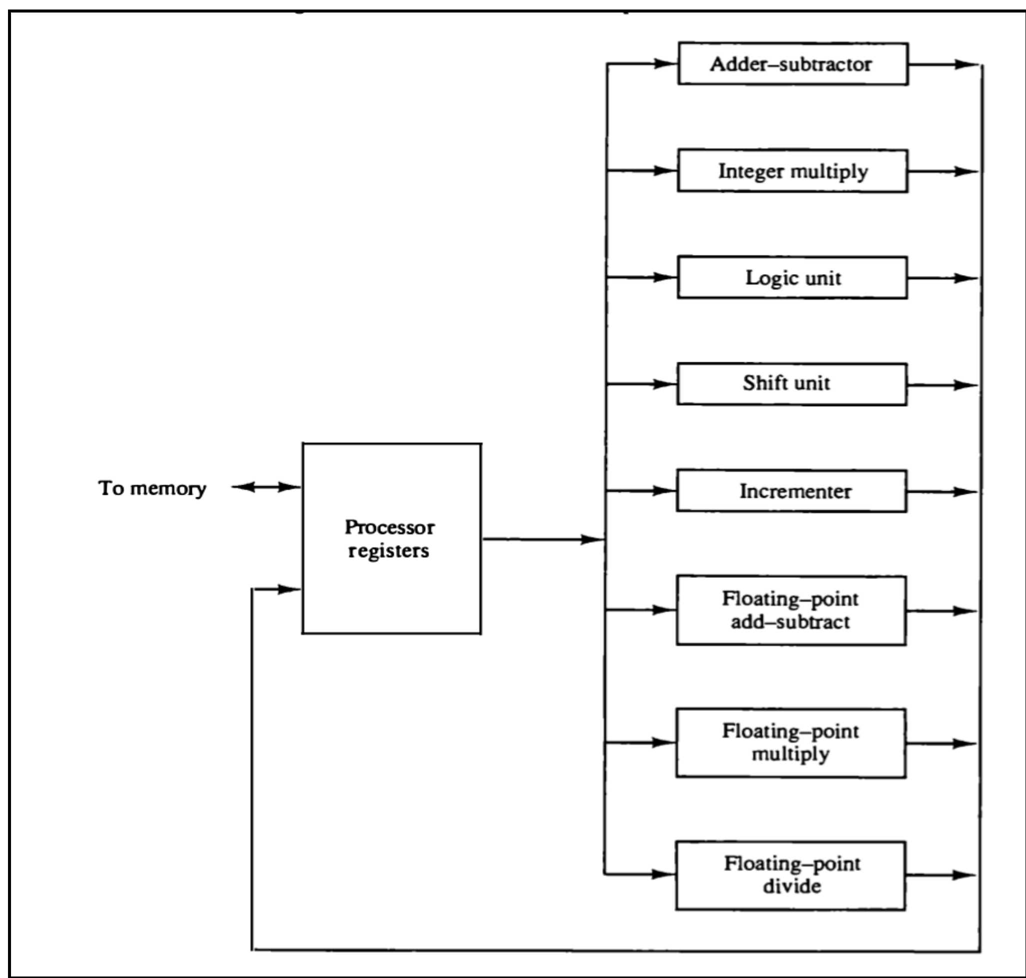


Figure 11.2 Processor with multiple functional units

(Source- *Computer System Architecture*, M. Morris Mano**)**

11.3 Pipelining

It is a technique of dividing a sequential process into sub-opera-tions, with each sub-process being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows.

Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.

The name "pipeline" implies a flow of information analogous to an industrial assembly line. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time. The information flows through the pipeline as per following steps, one step at a time.

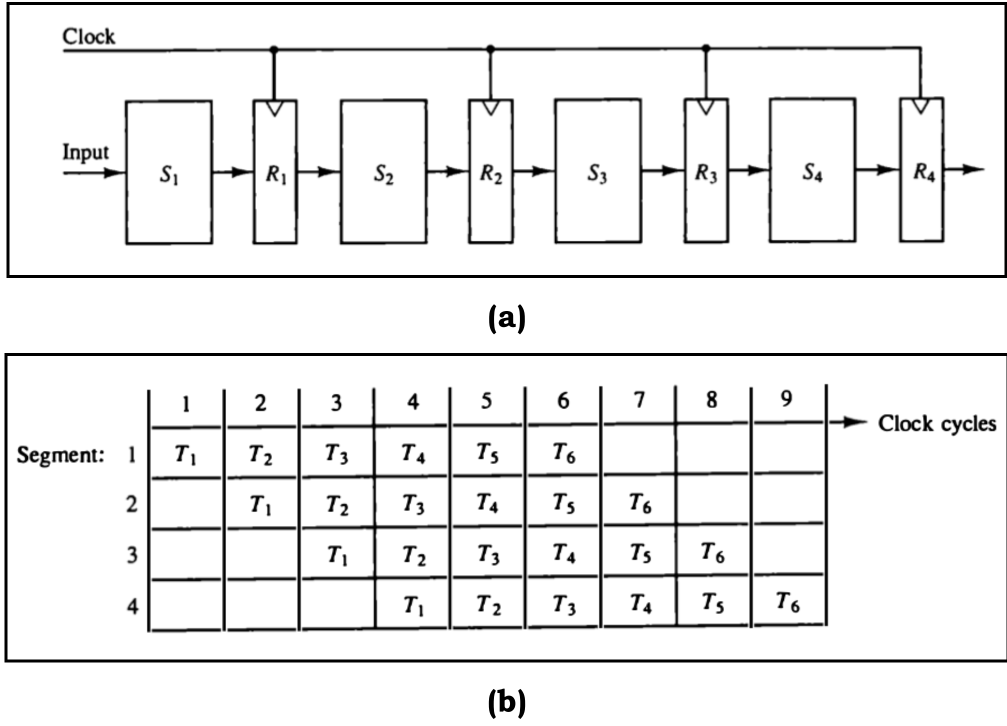
- The overlapping of computation is made possible by associ-ating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.
- Each segment consists of an input register followed by a combinational circuit. The register holds the data and the combinational circuit performs the sub-operation in the particular segment.
- The output of the combinational circuit in a given segment is applied to the input register of the next segment.
- A clock is applied to all registers after enough time has elapsed to perform all segment activity.

Any operation that can be decomposed into a sequence of sub-operations of about the same complexity can be implemented by a pipeline processor. The technique is efficient for those applications that need to repeat the same task many times with different sets of data. The behavior of a pipeline can be illustrated with a ***space-time diagram***. This is a diagram that shows the segment utilization as a function of time.

Figure 11.3 (a) illustrates the general structure of a four-segment pipeline. The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit S_i that performs a sub operation over the data stream flowing through the pipe. The segments are separated by registers R_i that hold the intermediate results between the stages. Information flows

between adjacent stages under the control of a common clock applied to all the registers simultaneously. We define a ***task*** as the total operation performed going through all the segments in the pipeline.

Figure 11.3 (b) demonstrates the space-time diagram of a four-segment pipeline. The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number. The diagram shows six tasks T_1 through T_6 executed in four segments. Initially, task T_1 is handled by segment 1. After the first clock, segment 2 is busy with T_2, while segment 1 is busy with task T_2. Continuing in this manner, the first task T_1 is completed after the fourth clock cycle. From then on, the pipe completes a task every clock cycle. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock period to obtain an output.



Segment:	1	2	3	4	5	6	7	8	9
1	T_1	T_2	T_3	T_4	T_5	T_6			
2		T_1	T_2	T_3	T_4	T_5	T_6		
3			T_1	T_2	T_3	T_4	T_5	T_6	
4				T_1	T_2	T_3	T_4	T_5	T_6

Figure 11.3 (a) Four-segment pipeline (b) Space-time diagram for the pipeline

Nevertheless, the pipeline technique provides a faster operation over a purely serial sequence even though the maximum theoretical speed is never fully achieved.

There are two areas of computer design where the pipeline organization is applicable. An ***arithmetic pipeline*** divides an arithmetic operation into sub--operations for execution in the pipeline segments. An ***instruction pipeline*** oper-ates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle.

Arithmetic Pipeline

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems. A pipeline multiplier is essentially an array multiplier with special adders designed to minimize the carry propagation time through the partial products. Floating-point operations are easily decomposed into sub-operations The inputs to the floating-point adder pipeline are two normalized float-ing-point binary numbers.

$$X = A \text{ X } 2^a$$

$$Y = B \text{ X } 2^b$$

A & B are two fractions that represent the mantissas a & b are the exponents. The floating-point addition and subtraction can be performed in four segments. The registers labeled R are placed between the segments to store intermediate results. The sub-operations that are per-formed in the four segments are:

- Compare the exponents.
- Align the mantissas.
- Add or subtract the mantissas.
- Normalize the result.

Instruction Pipeline

Pipeline processing can occur not only in the data stream but in the instruction stream as well. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations.

As a simple approach, consider subdividing instruction processing into two stages: ***fetch instruction*** and ***execute instruction***. There are times during the execution of an instruction when main memory is not being accessed. This time could be used to fetch the next instruction in parallel with the execution of the current one. This approach is demonstrated in figure 11.4 (a). The pipeline has two independent stages.

1. The first stage fetches an instruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction.
2. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This is called ***instruction prefetch*** or ***fetch overlap***.

Note that this approach, which involves instruction buffering, requires more registers. In general, pipelining requires registers to store data between stages. It should be clear that this process will speed up instruction execution. If the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (figure 11.4 (b), we will see that this doubling of execution rate is unlikely for two reasons:

- The execution time will generally be longer than the fetch time. Execution will involve reading and storing operands and the performance of some operation. Thus, the fetch stage may have to wait for some time before it can empty its buffer.
- A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

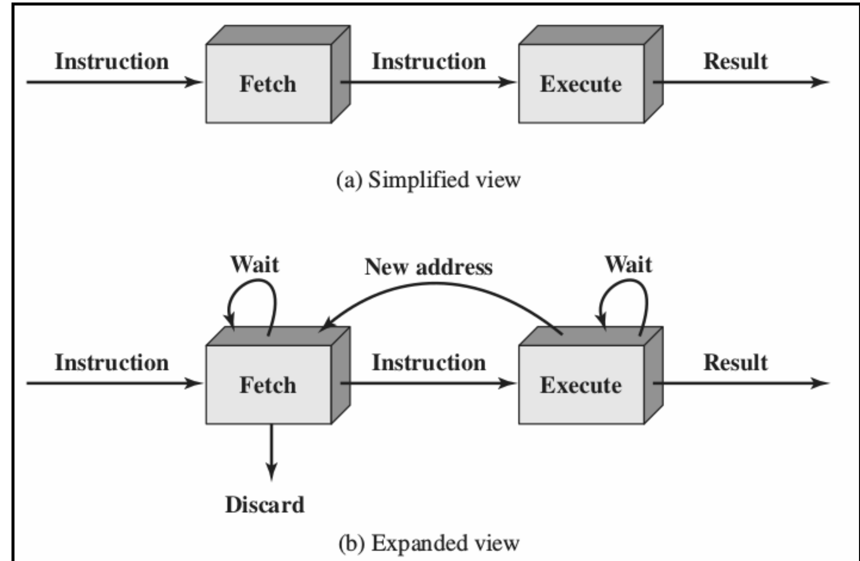


Figure 11.4 Two-Stage Instruction Pipeline

Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. In the most general case, the computer needs to process each instruction with the following se-quence of steps:

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

The design of an instruction pipeline will be most efficient if the instruc-tion cycle is divided into segments of equal duration. The time that each step takes to fulfill its function depends on the instruction and the way it is executed.

In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. ***Resource conflicts*** caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.

2. ***Data dependency*** conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
3. ***Branch difficulties*** arise from branch and other instructions that change the value of the PC .

11.4 Data Dependency

A difficulty that may cause a degradation of performance in an instruction pipeline is due to possible collision of data or address. A collision occurs when an instruction cannot proceed because previous instructions did not complete certain operations. A ***data dependency*** occurs when an instruction needs data that is not yet available. An ***address dependency*** may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available. Pipelined computers deal with such conflicts between data depen-dencies in a variety of ways.

- **Hardware Interlocks:** The most straightforward method is to insert hardware interlocks. An ***interlock*** is a circuit that detects instructions whose source operands are des-tinations of instructions farther up in the pipeline. Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delays.
- **Operand Forwarding:** Another technique called operand forwarding uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments. For example, instead of transferring an ALU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register ffie. This method requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.
- **Delayed Load:** A procedure employed in some computers is to give the responsibility for solving data conflicts problems to the compiler that

translates the high-level programming language into a machine language program. The compiler for such computers is designed to detect a data conflict and reorder the instruc-tions as necessary to delay the loading of the conflicting data by inserting no-operation instructions. This method is referred to as delayed load.

11.5 Handling of Branch Instructions

One of the major problems in designing an instruction pipeline is assuring a steady flow of instructions to the initial stages of the pipeline. The primary impediment, as we have seen, is the conditional branch instruction. Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not. A variety of approaches have been taken for dealing with conditional branches:

- **Multiple streams:** A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may make the wrong choice. A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams. There are two problems with this approach:
    a) With multiple pipelines there are contention delays for access to the registers and to memory.
    b) Additional branch instructions may enter the pipeline (either stream) before the original branch decision is resolved. Each such instruction needs an additional stream.

  Despite these drawbacks, this strategy can improve performance. Examples of machines with two or more pipeline streams are the IBM 370/168 and the IBM 3033.
- **Prefetch branch target:** When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is

executed. If the branch is taken, the target has already been prefetched. The IBM 360/91 uses this approach.

- **Loop buffer:** Loop buffer is a small, very high-speed register file maintained by the instruction fetch segment of the pipeline. When a program loop is detected in the program, it is stored in the loop buffer in its entirety, including all branches. The program loop can be executed directly without having to access memory until the loop mode is removed by the final branching out. The loop buffer has three benefits:
    a) With the use of prefetching, the loop buffer will contain some instruction sequentially ahead of the current instruction fetch address. Thus, instructions fetched in sequence will be available without the usual memory access time.
    b) If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer. This is useful for the rather common occurrence of If-Then and If-Then-Else sequences.
    c) This strategy is particularly well suited to dealing with loops, or iterations; hence the name loop buffer. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.
- **Branch prediction:** Another procedure that some computers use is branch prediction. A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins prefetching the instruction stream from the predicted path. A correct prediction eliminates the wasted time caused by branch penalties. Various techniques can be used to predict whether a branch will be taken. Among the more common are Predict never taken; Predict always taken; Predict by opcode; Taken/not taken switch; Branch history table.

- **Delayed branch:** A procedure employed in most ruse processors is the delayed branch. In this procedure, the compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions. It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired.

11.6 Vector Processing

High computational power is a never ending requirement. Certain classes of computational problems are beyond the ability of a conventional computer. The scientific and research computations involve many computations which require extensive and high power computers. These computations when run in a conventional computer may take days or weeks to complete. The science and engineering problems can be formulated in terms of vectors and matrices using ***vector processing***.

Computers with vector processing capabilities are in demand in specialized applications. Some representative application areas where vector processing is of the utmost importance are Long-range weather forecasting; Petroleum explorations; Seismic data analysis; Medical diagnosis; Aerodynamics and space flight simulations; Artificial intelligence and expert systems; Mapping the human genetic data; Image processing.

To achieve the required level of high performance it is necessary to utilize the fastest and most reliable hardware and apply innovative procedures from vector and parallel processing techniques.

Characteristics of Vector Processing

A vector is a structured set of elements. The elements in a vector are scalar quantities. A vector operand contains an ordered set of n elements, where n is called the length of the vector. Each clock period processes two successive pairs of elements. During one single clock period the dual vector pipes and the dual sets of vector functional units allow the processing of two pairs of

elements. As the completion of each pair of operations takes place, the results are delivered to appropriate elements of the result register. The operation continues until the number of elements processed is equal to the count specified by the vector length register.

In parallel vector processing, more than two results are generated per clock cycle. The parallel vector operations are automatically initiated under the following two circumstances:

- Firstly, when successive vector instructions use different functional units and different vector registers.
- Secondly, when successive vector instructions use the result stream from one vector register as the operand of another operation using a different functional unit. This process is termed as ***chaining***.

A vector processor performs better with longer vectors due to the startup delay in a pipeline. Vector processing reduces the overhead associated with maintenance of the loop-control variables which makes it more efficient than scalar processing.

Although vector processing is not for general purpose computation, it does offer significant advantages for scientific computing applications. The following are the advantages of using vector processing:

- Flynn's bottleneck can be reduced by using vector instructions as each vector instruction specifies a lot of work. Vector instructions can specify the work equivalent of an entire loop. Thus, fewer instructions are required to execute programs. This reduces the bandwidth required for instruction fetch. Flynn's bottleneck states that the maximum processor element (PE) can process only one instruction per clock cycle.
- Data hazards can be eliminated due to the structured nature of the data used by vector machines. We can determine the absence of a data hazard at compile-time, which not only improves performance but also allows for planned pre-fetching of data from memory.
- Memory latency can be reduced by using pipelined load and store operations.

- Control hazards are reduced as a result of specifying a large number of iterations in a single vector instruction. The number of iterations depends on the size of the vector registers.
- Pipelining can be exploited to the maximum extent. This is facilitated by the absence of data and control hazards. Vector machines not only use pipelining for integer and floating-point operations, but also to feed data from one functional unit to another. This process is known as chaining. In addition, as mentioned before, load and store operations also use pipelining.

Supercomputers

Supercomputers are very powerful, high-performance machines used mostly for scientific compu-tations. A commercial computer with vector instructions and pipelined floating-point arithmetic operations is referred to as a ***supercomputer***. To speed up the operation, the components are packed tightly together to minimize the distance that the electronic signals have to travel. Supercom-puters also use special techniques for removing the heat from circuits to prevent them from burning up because of their close proximity.

Supercomputers are not suitable for normal everyday processing of a typical computer installation. They are limited in their use to a number of scientific applications, such as numerical weather forecasting, seismic wave analysis, and space research. They have limited use and limited market be-cause of their high price.

The instruction set of supercomputers contains the standard data trans-fer, data manipulation, and program control instructions of conventional com-puters. This is augmented by instructions that process vectors and combina-tions of scalars and vectors.

11.7 Array Processors

There is another type of system that has been designed to address the need for vector computation, referred to as the ***array processor***. Although a supercomputer is optimized for vector computation, it is a general-purpose

computer, capable of handling scalar processing and general data processing tasks. Array processors do not include scalar processing; they are configured as peripheral devices by both mainframe and minicomputer users to run the vectorized portions of programs.

An array processor is a processor that performs computations on large arrays of data. The term is used to refer to two different types of processors.

- An ***attached array processor*** is an auxiliary processor attached to a general-purpose computer. It is intended to improve the performance of the host computer in specific numerical computation tasks.
- An ***SIMD array processor*** is a processor that has a single-instruction multiple-data organization. It manipulates vector instructions by means of multiple functional units responding to a common instruction.

Although both types of array processors manipulate vectors, their internal organization is different.

Attached Array Processor

An attached array processor is designed as a peripheral for a conventional host computer, and its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications. It achieves high performance by means of parallel processing with multiple functional units. It includes an arithmetic unit containing one or more pipelined floating- point adders and multipliers. The array processor can be programmed by the user to accommodate a variety of complex arithmetic problems.

Figure 11.5 illustrates the interconnection of an attached array processor to a host computer. The attached processor is a back-end machine driven by the host computer and the host computer is a general-purpose commercial com-puter.

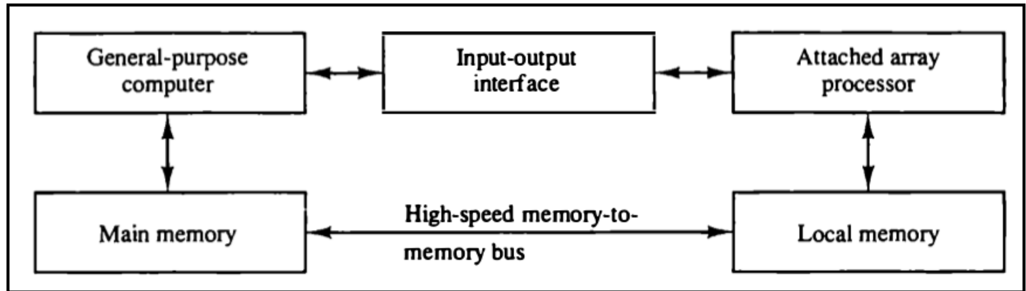


Figure 11.5 An Attached array Processor with host computer

The array processor is connected through an input-output con-troller to the computer and the computer treats it like an external interface. The data for the attached processor are transferred from main memory to a local memory through a high-speed bus. The general-purpose computer without the attached processor serves the users that need conventional data process-ing. The system with the attached processor satisfies the needs for complex arithmetic applications.

SIMD Array Processor

An SIMD array processor is a computer with multiple processing units oper-ating in parallel. The processing units are synchronized to perform the same operation under the control of a common control unit, thus providing a single instruction stream, multiple data stream (SIMD) organization. A general block diagram of an array processor is shown in figure 11.6. It contains a set of identical processing elements (PEs), each having a local memory M. Each processor element includes an ALU, a floating-point arithmetic unit, and working regis-ters. The master control unit controls the operations in the processor elements. The main memory is used for storage of the program. The function of the master control unit is to decode the instructions and determine how the instruction is to be executed. Scalar and program control instructions are directly executed within the master control unit. Vector instructions are broadcast to all PEs simultaneously. Each PE uses operands stored in its local

memory. Vector operands are distributed to the local memories prior to the parallel execution of the instruction.

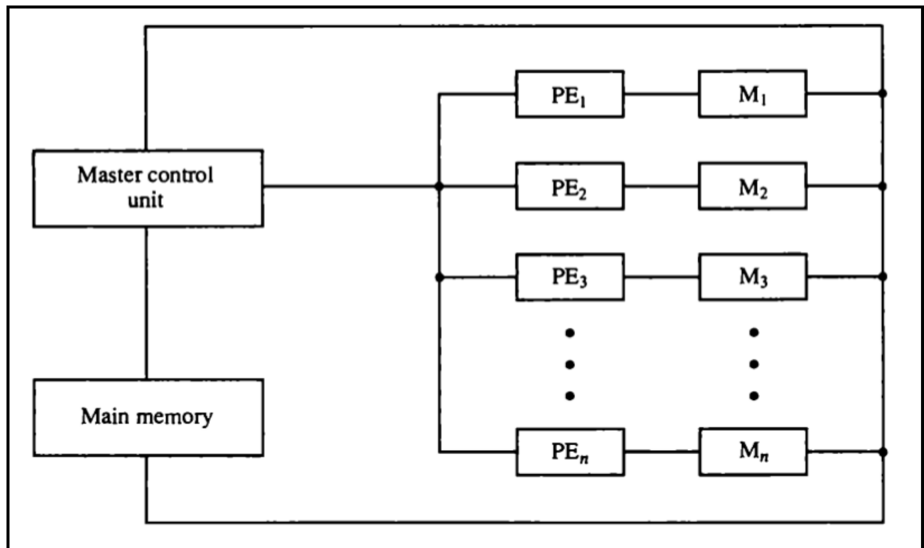


Figure 11.6 SIMD Array Processor organization

11.8 Summary

- Parallel processing is a technique of executing several jobs simultaneously in order to enhance the speed of processing and throughput.
- Pipelining is a technique of dividing a sequential process into sub operations, with each sub-process being executed in a special dedicated segment that operates concurrently with all other segments.
- An ***arithmetic pipeline*** divides an arithmetic operation into sub--operations for execution in the pipeline segments. An ***instruction pipeline*** oper-ates on a stream of instructions by overlapping the fetch, decode, and execute phases of the instruction cycle.
- There are three major pipelining conflicts: resource conflicts, data dependency conflicts and branch difficulties.
- The pipeline conflicts can be removed by these schemes: Hardware interlocks, operand forwarding, delayed load.

- Computers with vector processing capabilities are in demand to run specialized applications involving computations which are beyond the capabilities of a conventional computer.
- An array processor is a processor that performs computations on large arrays of data.

11.9 Key Terms

- ***Branch Penalty:*** The delay caused due to a branch instruction in a pipeline.
- ***Combinational Circuit:*** A combinational circuit is one for which the output value is determined solely by the values of the inputs.
- ***Latency:*** Measure of time delay that is experienced in a system.
- ***Chaining:*** Vector machines not only use pipelining for integer and floating-point operations, but also to feed data from one functional unit to another. This process is known as chaining.
- ***Delayed branch:*** A procedure employed in most ruse processors is the delayed branch.

11.10 Check Your Progress

Q1) Write a short note on Pipelining in parallel processing.

Q2) What is an array processor? Discuss its types.

Q3) List the advantages of Vector Processing.

Q4) What are the three major pipelining conflicts?

Q5) Define Loop buffer. How is it used to remove the pipelining conflicts?

References:

Computer Organization and Architecture, 9th edition, William Stallings, Pearson Publication.

Computer System Architecture, M. Morris Mano

The essentials of Computer Organization and Architecture, Linda Null and Julia Lobur, Jones & Bartlett Learning.

Unit 12 – Multiprocessors

Structure

12.0 Introduction
12.1 Unit Objectives
12.2 Characteristics of Multiprocessors
12.3 Types of Multiprocessors
12.4 Interconnection Structures
12.4.1 Time-Shared Common Bus
12.4.2 Multiport Memory
12.4.3 Crossbar switch
12.4.4 Multistage switching network
12.4.5 Hypercube system
12.5 Interprocessor Arbitration
12.5.1 Serial Arbitration Procedure
12.5.2 Parallel Arbitration Logic
12.5.3 Dynamic Arbitration Algorithms
12.6 Inter-Processor Communication And Synchronization
12.7 Symmetric Multiprocessors
12.8 Summary
12.9 Key Terms
12.10 Check Your Progress

12.0 Introduction

Multiprocessor is a single computer that has multiple processors. It is possible that the processors in the multiprocessor system can communicate and cooperate at various levels of solving a given problem. The communications between the processors take place by sending messages from one processor to another, or by sharing a common memory.

Both multiprocessors and multicomputer systems share the same fundamental goal, which is to perform the concurrent operations in the system. However, there is a significant difference between multicomputer systems and multiprocessors. The difference exists depending on the extent of resource sharing and cooperation in solving a problem. A multicomputer system

includes numerous autonomous computers which may or may not communicate with each other. However, a single operating system that provides communication between processors and their programs on the process, data set, and data element level, controls a multiprocessor system.

The interprocessor communication is carried out with the help of shared memories or through an interrupt network. Most significantly, a single operating system that provides interactions between processors and their programs at different levels, controls the whole system.

12.1 Unit Objectives

On completion of this unit, the reader will be able to:

- Describe the characteristics of multiprocessor
- Discuss the uses of multiprocessors
- Explain interconnection structures
- Illustrate interprocessor communication and synchronization

12.2 Characteristics of Multiprocessors

Multiprocessor is a data processing system that can execute more than one program or more than one arithmetic operation simultaneously. It is also known as a multiprocessing system. Multiprocessor uses more than one processor and is similar to multiprogramming that allows multiple threads to be used for a single procedure. The term 'multiprocessor' can also be used to describe several separate computers running together. It is also referred to as clustering. A system is called a multiprocessor system only if it includes two or more elements that can implement instructions independently. A multiprocessor system employs a distributed approach. In a distributed approach, a single processor does not perform a complete task. Instead more than one processor is used to do the subtasks.

Some of the major characteristics of multiprocessors include:

- ***Parallel Computing:*** This involves simultaneous application of multiple processors. These processors are developed using a single architecture in order to execute a common task. In general, processors are identical and they work together in such a way that the users are under the impression that they are the only users of the system. In reality, however, there are many users accessing the system at a given time.
- ***Distributed Computing:*** This involves the usage of a network of processors. Each processor in this network can be considered as a computer in its own right and have the capability to solve a problem. These processors are heterogeneous, and generally one task is allocated to a single processor.
- ***Supercomputing:*** This involves usage of the fastest machines to resolve big and computationally complex problems. In the past, supercomputing machines were vector computers but at present, vector or parallel computing is accepted by most people.
- ***Pipelining:*** This is a method wherein a specific task is divided into several subtasks that must be performed in a sequence. The functional units help in performing each subtask. The units are attached in a serial fashion and all the units work simultaneously.
- ***Vector Computing:*** It involves usage of vector processors, wherein operations such as 'multiplication' is divided into many steps and is then applied to a stream of operands ("vectors").
- ***Systolic:*** This is similar to pipelining, but units are not arranged in a linear order. The steps in systolic are normally small and more in number and performed in a lockstep manner. This is more frequently applied in special-purpose hardware such as image or signal processors.

A multiprocessor system has the following advantages:

- It helps to improve the cost or performance ratio of the system.
- It helps to fit the needs of an application, when several processors are combined. At the same time, a multiprocessor system avoids the

expenses of the unnecessary capabilities of a centralized system. However, this system provides room for expansion.

- It helps to divide the tasks among the modules. If failure happens, it is simple and cheap to identify and replace the malfunctioning processor, instead of replacing the failing part of the complex processor.
- It helps to improve the reliability of the system. A failure that occurs in any one part of a multiprocessor system has a limited effect on the rest of the system. If an error occurs in one processor, a second processor may take up the responsibility of doing the task of the processor in which the error has occurred. This helps in enhancing the reliability of the system at the cost of some loss in efficiency.

12.3 Types of Multiprocessors

Multiprocessors are classified by the way their memory is organized. There are two types of multiprocessor systems:

1. **Tightly-coupled Multiprocessor System:**

   A multiprocessor system with common shared memory is classified as a ***shared- memory or tightly coupled multiprocessor.*** This does not preclude each processor from having its own local memory. In fact, most commercial tightly coupled multiprocessors provide a cache memory with each CPU. In addition, there is a global common memory that all CPUs can access. Information can therefore be shared among the CPUs by placing it in the common global memory. This system has many CPUs that are attached at the bus level. Tasks and/or processors interact in a highly synchronized manner. The CPUs have access to a central shared memory and communicate through a common shared memory.

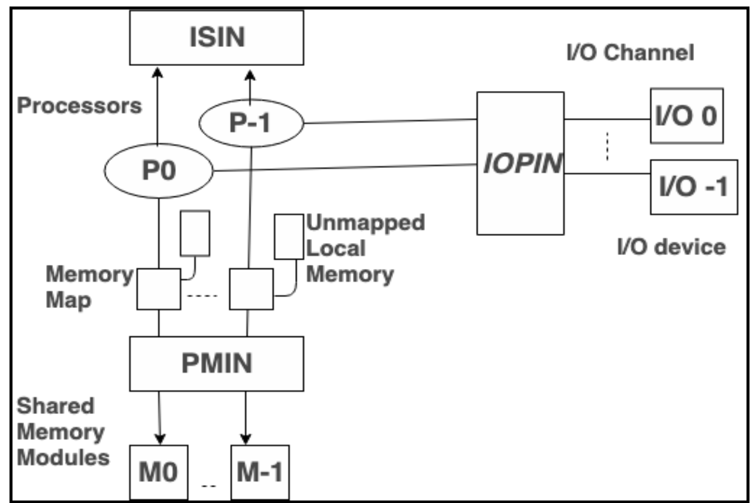


Figure 12.1 Tightly Coupled Multiprocessor System

2. **Loosely-coupled Multiprocessor System:**

An alternative model of microprocessor is the ***distributed-memory or loosely coupled system.*** Each processor element in a loosely coupled system has its own private local memory. The processors are tied together by a switching scheme designed to route information from one processor to another through a mes-sage-passing scheme. The processors relay program and data to other proces-sors in packets. A packet consists of an address, the data content, and some error detection code. The packets are addressed to a specific processor or taken by the first available processor, depending on the communication system used. Loosely coupled systems are most efficient when the interaction between tasks is minimal, whereas tightly coupled systems can tolerate a higher degree of interaction between tasks.

This multiprocessor system is often referred to as ***clusters***. These systems operate based on single or dual processor commodity computers interconnected through a high speed communication system. Tasks or processors do not communicate in a synchronized manner as done in tightly-coupled multiprocessor systems. They communicate through the

Message Transfer System. This system has a high overhead for data exchange and uses a distributed memory system.

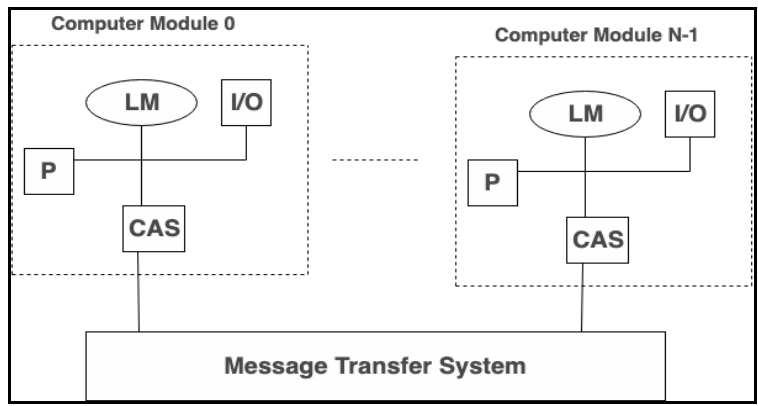


Figure 12.2 Loosely Coupled Multiprocessor System

Table 12.1 Difference between Loosely Coupled and Tightly Coupled Multiprocessors

S.NO.	Loosely Coupled Multiprocessors	Tightly Coupled Multiprocessors
1.	There is distributed memory in a loosely coupled multiprocessor system.	There is shared memory, in a tightly coupled multiprocessor system.
2.	Loosely Coupled Multiprocessor System has low data rate.	Tightly coupled multiprocessor system has a high data rate.
3.	The cost of a loosely coupled multiprocessor system is less.	Tightly coupled multiprocessor system is more costly.
4.	In a loosely coupled multiprocessor system, modules are connected through a Message transfer system network.	While there is PMIN, IOPIN and ISIN networks.

5.	In a loosely coupled multiprocessor, Memory conflicts don’t take place.	While tightly coupled multiprocessor systems have memory conflicts.
6.	A Loosely Coupled Multiprocessor system has a low degree of interaction between tasks.	Tightly Coupled multiprocessor system has a high degree of interaction between tasks.
7.	In a loosely coupled multiprocessor, there is direct connection between processor and I/O devices.	While in a tightly coupled multiprocessor, IOPIN helps connection between processor and I/O devices.
8.	Applications of loosely coupled multiprocessors are in distributed computing systems.	Applications of tightly coupled multiprocessors are in parallel processing systems.

12.4 Interconnection Structures

The structures that are used to connect the memories and processors (and between memories and I/O channels if required), are called interconnection structures. A multiprocessor system is formed by elements such as CPUs, peripherals, and a memory unit that is divided into numerous separate modules. There can exist different physical configurations for the interconnection between the elements. The physical configurations are based on the number of transfer paths existing between the processors and memory in a shared memory system or among the processing elements in a loosely coupled system. An interconnection network is established using several physical forms available. Some of the physical forms include:

- Time-shared common bus
- Multiport memory

- Crossbar switch
- Multistage switching network
- Hypercube system

12.4.1 Time-Shared Common Bus

In time-shared common bus, there are numerous processors connected through a common path to the memory unit in a common-bus multiprocessor system. Figure 12.3 shows organization of time-shared common bus for five processors. At any specified time, only one processor can communicate with the memory or another processor. The processor that is in control of the bus at the time performs transfer operations. Any processor that wants to initiate a transfer must first verify the availability status of the bus.

Once the bus is available, the processor can establish a connection with the destination unit to initiate the transfer. A command is issued to inform the destination unit about the function to be performed. The receiving unit identifies its address in the bus, and then responds to the control signals from the sender, after which the transfer is initiated. As all processors share a common bus, it is possible that the system may display some transfer conflicts. Incorporation of a bus controller that creates priorities among the requesting units helps in resolving the transfer conflicts.

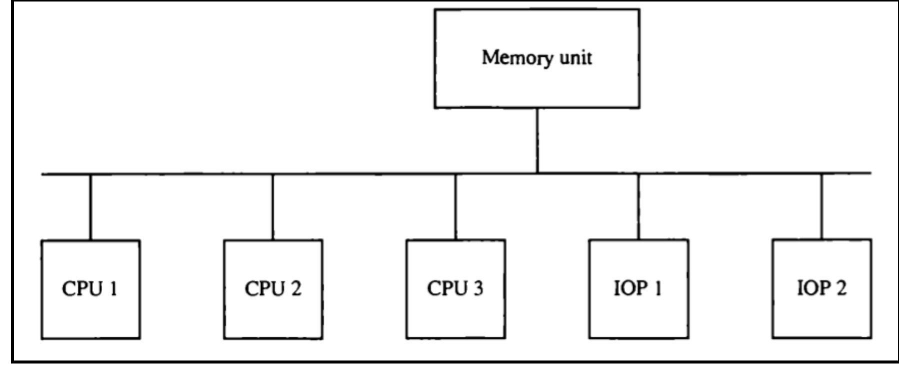


Figure 12.3 Organization of a Time-Shared Common Bus

There is a restriction of one transfer at a time for a single common-bus system.

This means that other processors are busy with internal operations or remain idle waiting for the bus when one processor is communicating with the memory. Hence, the speed of the single path limits the total overall transfer rate within the system. The system processors are kept busy through the execution of two or more independent buses, to allow multiple bus transfers simultaneously. However, this leads to an increase in the system cost and complexity.

Figure 12.4 depicts a more economical execution of a dual bus structure for multiprocessors.

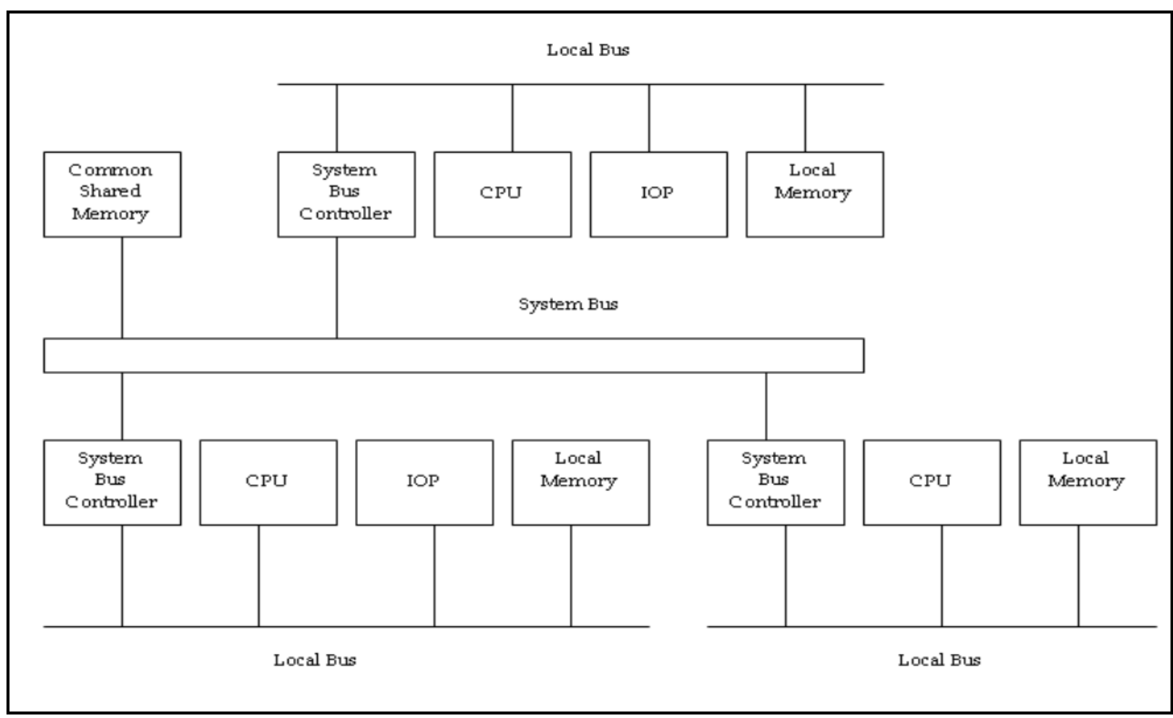


Figure 12.4 System bus structure for Multiprocessors

In figure 12.4, we see that there are many local buses, and each bus is connected to its own local memory, and to one or more processors. Each local bus is connected to a peripheral, a CPU, or any combination of processors. Each local bus is linked to a common system bus using a system bus controller.

The I/O devices connected to both the local I/O peripherals and the local memory are available to the local processor. All processors share the memory

connected to the common system bus. When an IOP is connected directly to the system bus, the Input/Output devices attached to it are made available to all processors. At any specified time, only one processor can communicate with the shared memory, and other common resources through the system bus. All the other processors are busy communicating with their local memory and I/O devices.

12.4.2 Multiport Memory

Multiport memory is a memory that helps in providing more than one access port to separate processors or to separate parts of one processor. A bus can be used to achieve this kind of access. This mechanism is applicable to interconnected computers too. A multiport memory system uses separate buses between each CPU and each memory module. Figure 12.5 depicts a multiport memory system for four CPUs and four ***Memory Modules (MMs)***. Every processor bus is connected to each memory module.

A processor bus consists of three elements; namely: ***address, data, and control lines***. These elements are needed to communicate with memory. Memory module has four ports and each port contains one of the buses. It is necessary for a module to have internal control logic to verify which port will have access to memory at any specified time. Assigning fixed priorities to each memory port helps in resolving memory access conflicts. The priority for memory access related to each processor is created with the physical port position that its bus occupies in each module. Consequently, CPU1 has priority over CPU2, CPU2 has priority over CPU3, and CPU4 has the least priority.

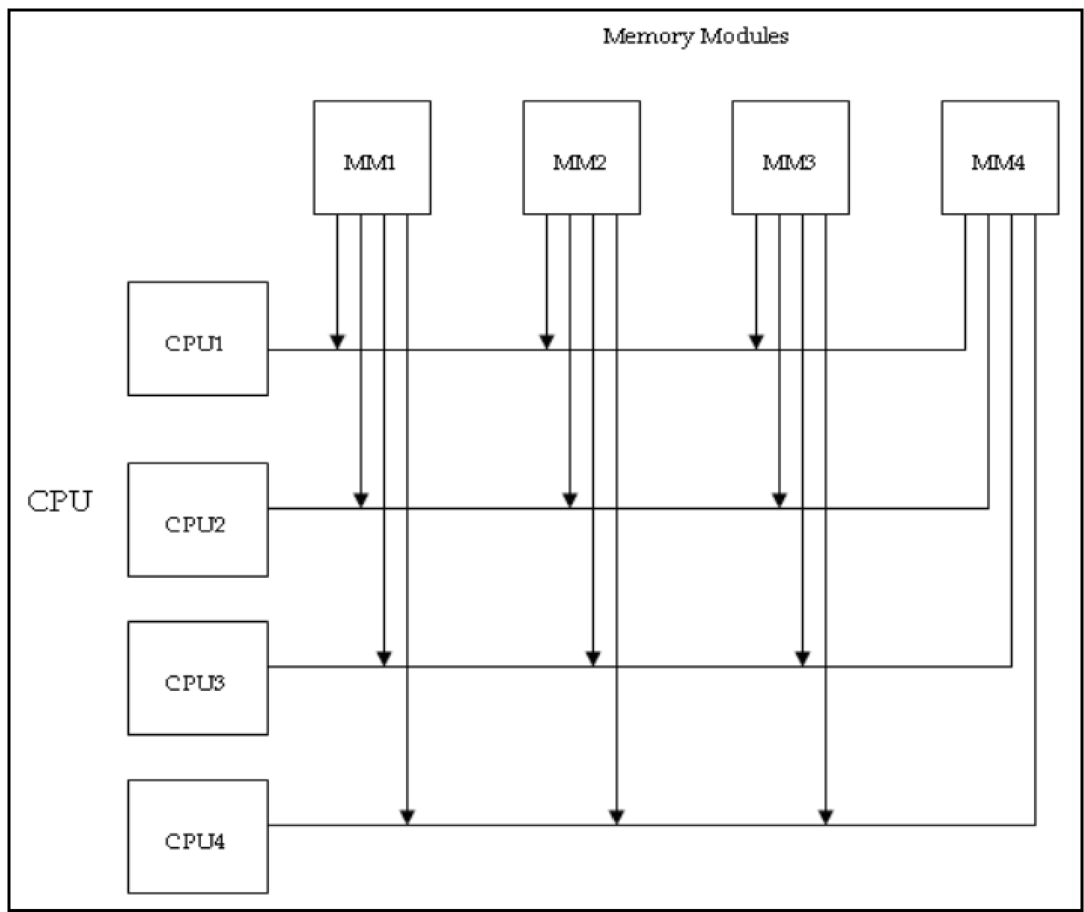


Figure 12.5 Multiport Memory Organization

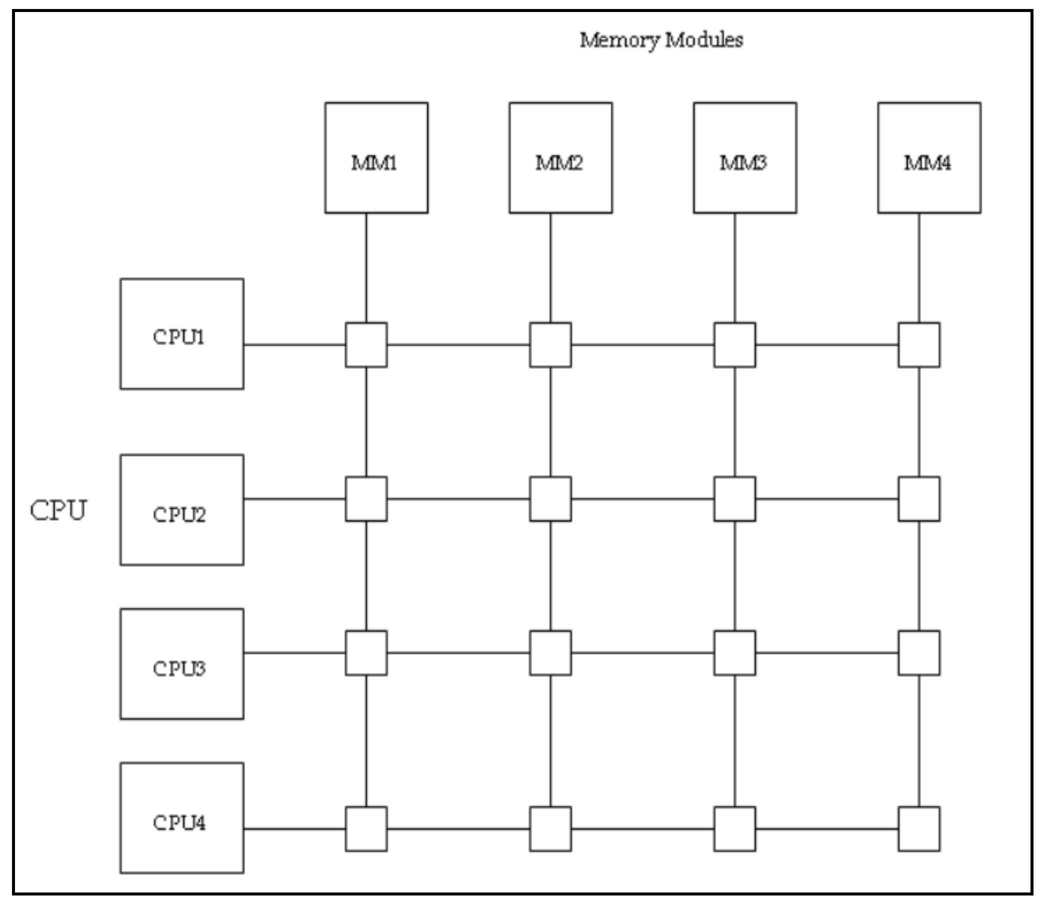
The multiport memory organization has an advantage of high transfer rate. This is because of several paths between memory and processors. The only drawback is that it needs expensive memory control logic and more cables and connectors. Therefore, this interconnection structure is usually suitable for systems having a small number of processors.

12.4.3 Crossbar Switch

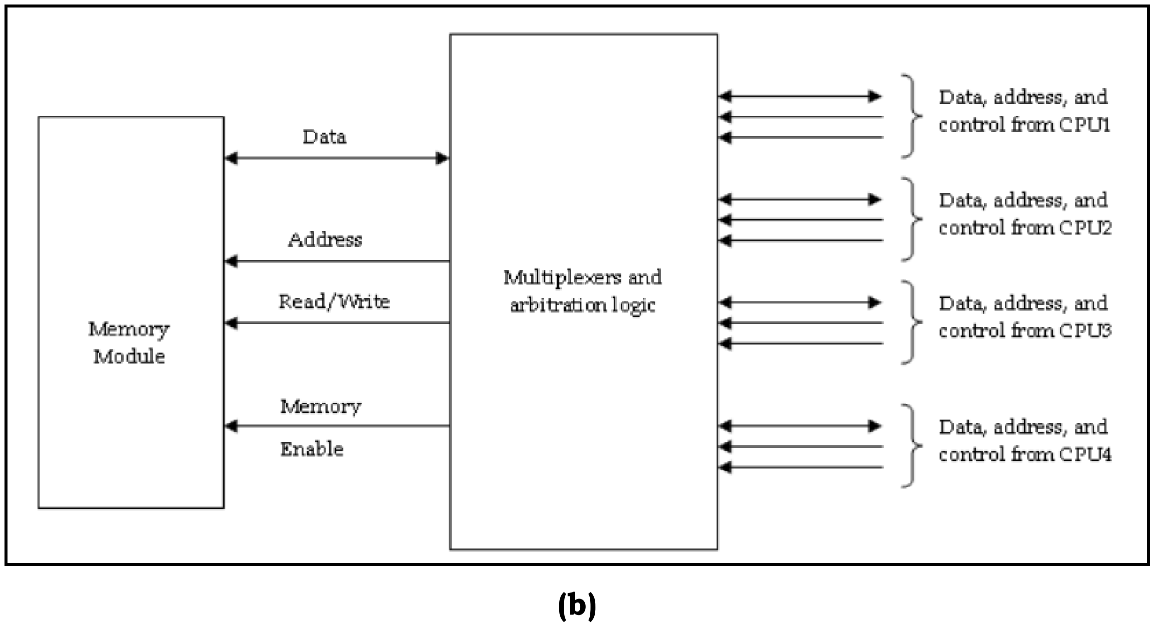
In a network, a device that helps in channeling data between any two devices that are connected to it, up to its highest number of ports is a crossbar switch. The paths set up between devices can be fixed for some period of time or changed when wanted.

In a crossbar switch organization, there are several cross points that are kept

at intersections between processor buses and memory module paths. Figure 12.6 (a) demonstrates a crossbar switch interconnection between four memory modules and four CPUs. The functional design of a crossbar switch connected to one memory module is depicted in figure 12.6 (b).



(a)



(b)

Figure 12.6 (a) Crossbar Switch (b) Block Diagram of a Crossbar Switch

In figure 12.6 (a), the small square in each crosspoint indicates a switch. This switch determines the path starting from a processor to a memory module. There is control logic for each switch point to set up the transfer path between a memory module and a processor. It checks the address that is placed in the bus to verify if its particular module is addressed. It also allows resolving multiple requests to get access to the same memory module on a predetermined priority basis.

In figure 12.6 (b), the circuit includes multiplexers that choose the data, address, and control from one CPU for communication with the memory module. The arbitration logic establishes priority levels to choose one CPU when two or more CPUs try to get access to the same memory. The binary code controls the multiplexers. A priority encoder generates this binary code within the arbitration logic.

12.4.4 Multistage Switching Network

The network that is built from small (for example, 2 x 2 crossbar) switch nodes along with a regular interconnection pattern is a multistage switching network.

Two-input, two-output interchange switch is a fundamental element of a multistage network. There are two inputs marked A and B, and two outputs marked 0 and 1 in the 2 x 2 switch as shown in figure 12.7.

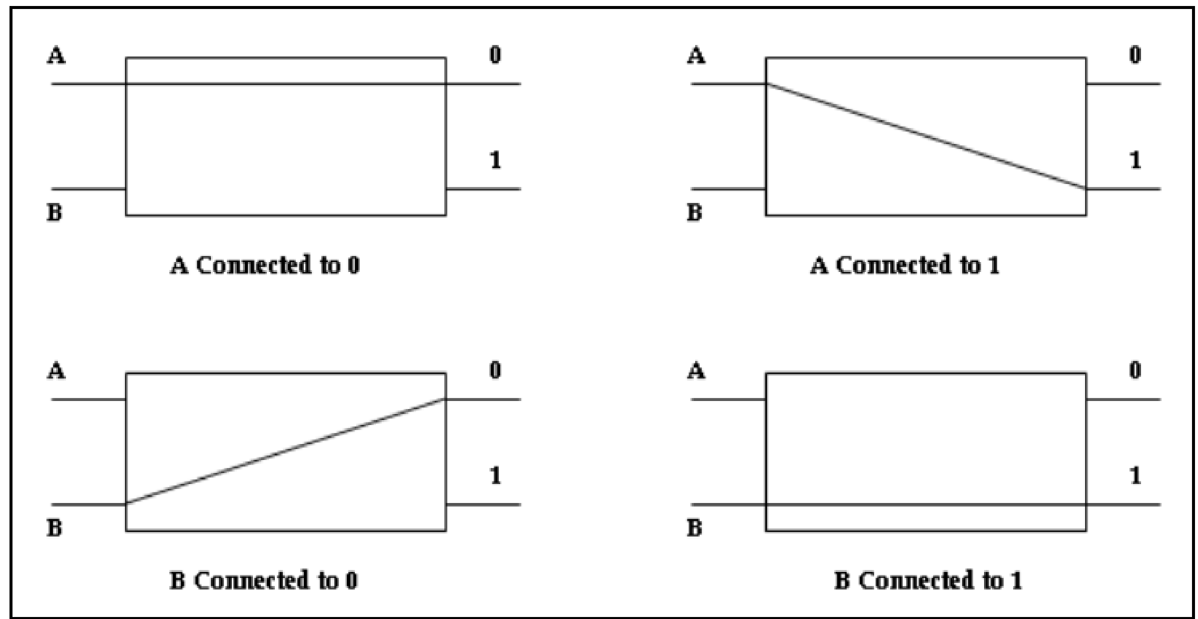


Figure 12.7 Operation of a 2 x 2 Interchange Switch

As depicted in the above diagram, there are control signals associated with the switch. The control signals establish interconnection between the input and output terminals. The switch can connect input A to either of the outputs. Terminal B of the switch acts in the same way. The switch can also arbitrate between conflicting requests. In case, inputs A and B request the same output terminals, it is possible that only one of the inputs is connected and the other is blocked.

It is possible to establish a multistage network to control the communication between numerous sources and destinations. The multistage network is established with the help of 2 x 2 switch as a building block. Consider the binary tree shown in figure 12.8 to see how this is carried out.

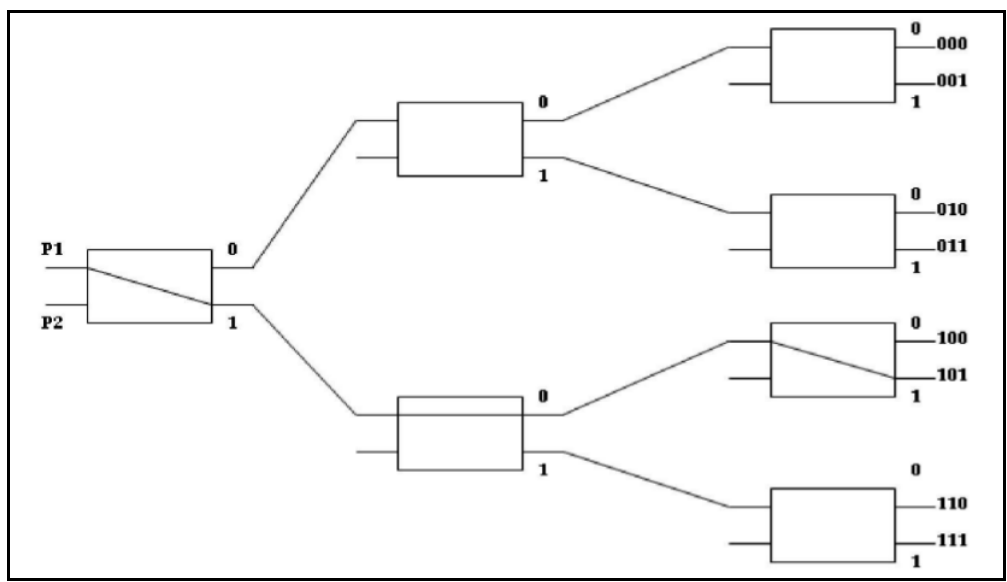


Figure 12.8 Binary Tree with 2 x 2 Switches

The two processors P1 and P2 are linked through switches to eight memory modules labeled in binary, starting from 000 through 111. The path starting from source to destination is determined from the binary bits of destination number. The first bit of the destination number helps in indicating the first level's switch output. The second bit identifies the second level's switch output, and the third bit specifies the third level's switch output.

As shown in figure 12.8, in order to make a connection between P1 and memory 101, it is important to create a path from P1 to output 1 in the third-level switch, output 0 in the second-level switch, and output 1 in the third-level switch. Hence, it is evident that either P1 or P2 must be connected to any one of the eight memories. If P1 is connected to one of the destinations 000 through 011, then it is possible to connect P2 to only one of the destinations 100 through 111.

Many different topologies have been proposed for multistage switching networks to control processor-memory communication in a tightly coupled multiprocessor system or to control the communication between the processing elements in a loosely coupled system. One such topology is the ***omega switching network*** shown in figure 11.9. In this configuration, there is exactly

one path from each source to any particular destination. Some request patterns, however, cannot be connected simultaneously. For example, any two sources cannot be connected simultaneously to destinations 000 and 001.

As depicted in figure 12.9, a specific request is started in the switching network through the source that sends a 3-bit pattern depicting the destination number. Every level checks a different bit to determine the 2 x 2 switch setting as the binary pattern moves through the network. Level 1 examines the most important bit, level 2 examines the middle bit, and level 3 examines the least important bit. When the request appears on input 2 x 2 switch, it is routed to the lower output if the specified bit is 1 or to the upper output if the specified bit is 0.

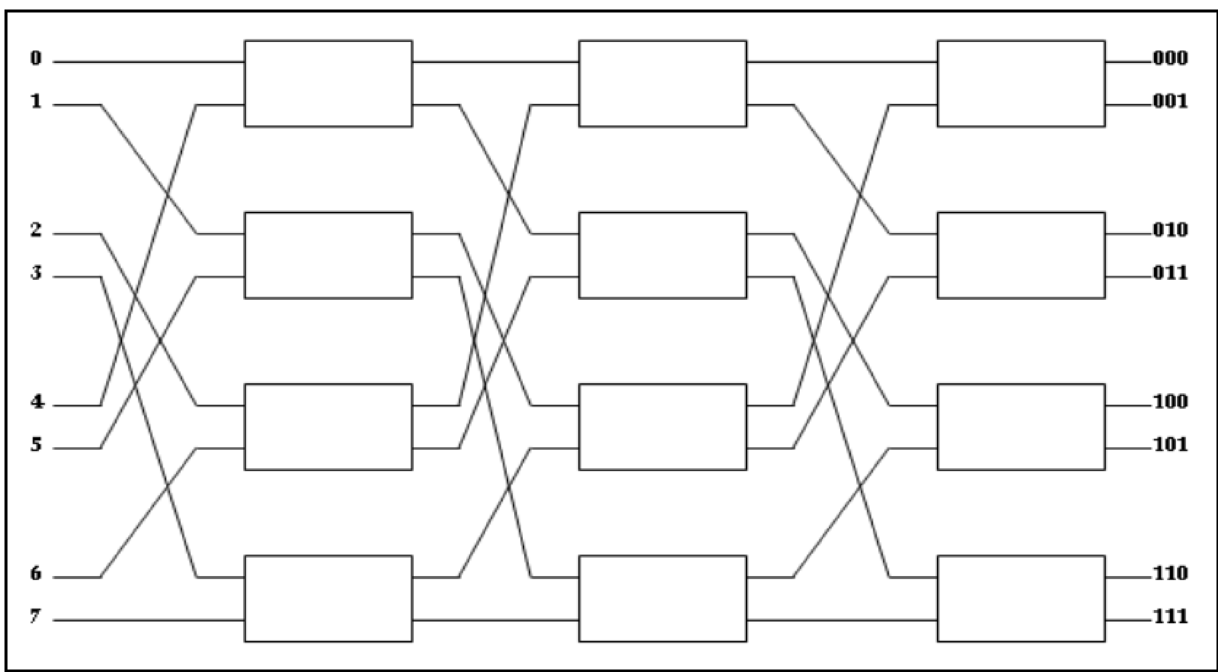


Figure 12.9 8 x 8 Omega Switching Network

The source is considered to be a processor and the destination is considered as a memory module in a tightly-coupled multiprocessor system. The path is set when the first pass is through the network. If the request is read or write the address is transferred into memory, and then the data is transferred in either direction using the succeeding passes. Both the destination and the source are

considered to be processing elements in a loosely-coupled multiprocessor system. The source processor transfers a message to the destination processor once the path is established.

11.4.5 Hypercube Interconnection

The hypercube is considered to be a loosely coupled system. The hypercube interconnection is also referred to as a ***binary n-cube multiprocessor***. This system is composed of $N = 2^n$ processors that are interconnected in an n-dimensional binary cube. Each processor indicates a node of the cube. Although it is expected to refer to every node as having a processor, in effect it not only has a CPU but also local memory and I/O interface. Every processor contains direct communication paths to n other neighbor processors. These paths relate to the edges of the cube. The processors can be assigned with 2^n distinct n-bit binary addresses. Each processor address differs from that of each of its n neighbors by exactly one bit position.

Figure 12.10 depicts the hypercube structure for n, wherein n = 1, 2, and 3. A one-cube structure contains n = 1 and $2^n = 2$. It has two processors that are interconnected by a single path. A two-cube structure contains n = 2 and $2^n = 4$. It has four nodes that are interconnected as a square. There are eight nodes interconnected as a cube in a three-cube structure. There are 2^n nodes in an n-cube structure with a processor existing in every node.

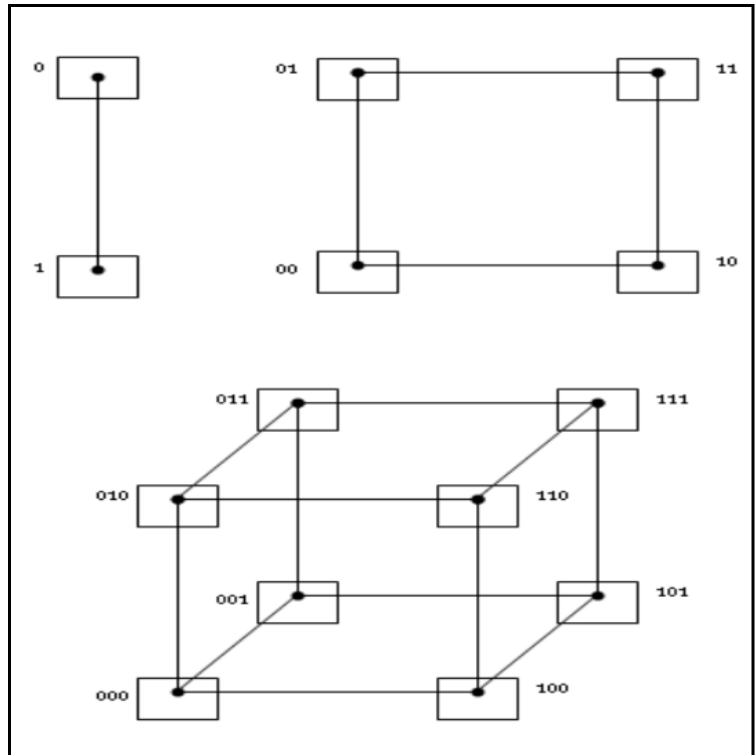


Figure 12.10 Hypercube Structure for n = 1, 2, and 3

Each node is assigned a binary address in such a way that the addresses of two neighbors differ in exactly one bit position. For example, the three neighbors of the node with address 100 in a three-cube structure are 000, 110, and 101. Each of these binary numbers differs from address 100 by one bit value.

Routing messages through an n-cube structure may take from one to n links from a source node to a destination node. For example, in a three-cube structure, node 000 can communicate directly with node 001. It must cross at least two links to communicate with 011 (from 000 to 001 to 011 or from 000 to 010 to 011). It is necessary to go through at least three links to communicate from node 000 to node 111.

A routing procedure can be developed by comput-ing the exclusive-OR of the source node address with the destination node address. The resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ. The message is then sent along any one of the axes. For example, in a three-

cube structure, a message at 010 going to 001 produces an exclusive-OR of the two addresses equal to 011. The message can be sent along the second axis to 000 and then through the third axis to 001.

The Intel iPSC has 128 (n = 7) microcomputers connected through communication channels. Each node has a CPU, local memory, floating-point processor, and serial communication interface units. The individual nodes work independently on data saved in local memory according to the resident programs. It is evident that the programs and data at every node is received through a message-passing system from other nodes or from a cube manager. Application programs are developed and gathered on the cube manager and then downloaded to the individual nodes. Computations are allocated through the system and implemented concurrently.

12.5 Interprocessor Arbitration

Computer systems contain a number of buses at various levels to facilitate the transfer of information between components. The CPU contains a number of internal buses for transferring information between processor registers and ALU. A memory bus consists of lines for transferring data, address, and read/write information. An I/O bus is used to transfer information to and from input and output devices. A bus that connects major components in a multi-processor system, such as CPUs, lOPs, and memory, is called a ***system bus***.

The processors in a shared memory multiprocessor system request access to common memory or other common resources through the system bus. If no other processor is currently utilizing the bus, the requesting processor may be granted access immediately. However, the requesting processor must wait if another processor is currently utilizing the system bus. Furthermore, other processors may request the system bus at the same time. Arbitration must then be performed to resolve this multiple contention for the shared resources. The arbitration logic would be part of the system bus controller placed between the local bus and the system bus as shown in figure 12.4.

System Bus

It is a typical system bus consisting of approximately 100 signal lines. These lines are divided into three functional groups: ***data, address, and control***. In addition, there are power distribution lines that supply power to the components. The data lines provide a path for the transfer o f data between processors and common memory. The number of data lines is usually a multiple of 8, with 16 and 32 being most common. The address lines are used to identify a memory address or any other source or destination, such as input or output ports. The number of address lines determines the maximum possible memory capacity in the system.

Data transfers over the system bus may be ***synchronous or asynchronous***.

- In a synchronous bus, each data item is transferred during a time slice known in advance to both source and destination units. Synchronization is achieved by driving both units from a common clock source. An alternative procedure is to have separate clocks of approximately the same frequency in each unit. Synchronization signals are transmitted periodically in order to keep all clocks in the system in step with each other.
- In an asynchronous bus, each data item being transferred is accompanied by handshaking control signals to indicate when the data are transferred from the source and received by the destination.

The ***six bus arbitration*** signals are used for interprocessor arbitration. These signals are explained below with the serial and parallel arbitration procedures.

12.5.1 Serial Arbitration Procedure

A hardware bus priority resolving technique can be established by means of a serial or parallel connection of the units requesting control of the system bus. The serial priority resolving technique is obtained from a ***daisy-chain connection*** of bus arbitration circuits similar to the priority inter-rupt logic. The processors connected to the system bus are assigned priority according to

their position along the priority control line. The device closest to the priority line is assigned the highest priority. When multiple devices concurrently request the use of the bus, the device with the highest priority is granted access to it.

Figure 12.11 depicts the daisy-chain connection of four arbiters. It is assumed that each processor has its own bus arbiter logic with priority-in and priority-out lines. The priority out (PO) of each arbiter is connected to the priority in (PI) of the next-lower-priority arbiter. The PI of the highest-priority unit is maintained at a logic 1 value. The highest-priority unit in the system will always receive access to the system bus when it requests it. The PO output for a particular arbiter is equal to 1 if its PI input is equal to 1 and the processor associated with the arbiter logic is not requesting control of the bus. This is the way that priority is passed to the next unit in the chain. If the processor requests control of the bus and the corresponding arbiter finds its PI input equal to 1, it sets its PO output to 0. Lower-priority arbiters receive a 0 in PI and generate a 0 in PO . Thus the processor whose arbiter has a PI = 1 and PO = 0 is the one that is given control of the system bus.

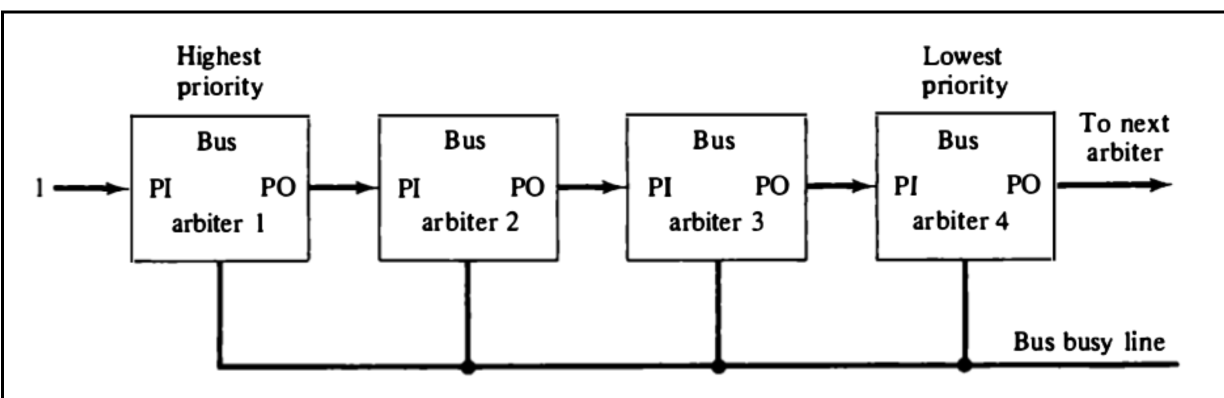


Figure 12.11 Serial (daisy-chain) arbitration.

A processor may be in the middle of a bus operation when a higher- priority processor requests the bus. The lower-priority processor must com-plete its bus operation before it relinquishes control of the bus. The bus busy line

shown in Fig. 13-10 provides a mechanism for an orderly transfer of control. The busy line comes from open-collector circuits in each unit and provides a wired-OR logic connection. When an arbiter receives control of the bus (because its PI = 1 and PO = 0) it examines the busy line. If the line is inactive, it means that no other processor is using the bus. The arbiter activates the busy line and its processor takes control of the bus. However, if the arbiter finds the busy line active, it means that another processor is currently using the bus. The arbiter keeps examining the busy line while the lower-priority processor that lost control of the bus completes its operation. When the bus busy line returns to its inactive state, the higher-priority arbiter enables the busy line, and its corresponding processor can then conduct the required bus transfers.

12.5.2 Parallel Arbitration Logic

The parallel bus arbitration technique uses an external priority encoder and a decoder as shown in figure 12.12. Each bus arbiter in the parallel scheme has a bus request output line and a bus acknowledge input line. Each arbiter enables the request line when its processor is requesting access to the system bus. The processor takes control of the bus if its acknowledged input line is enabled. The bus busy line provides an orderly transfer of control, as in the daisy-chaining case.

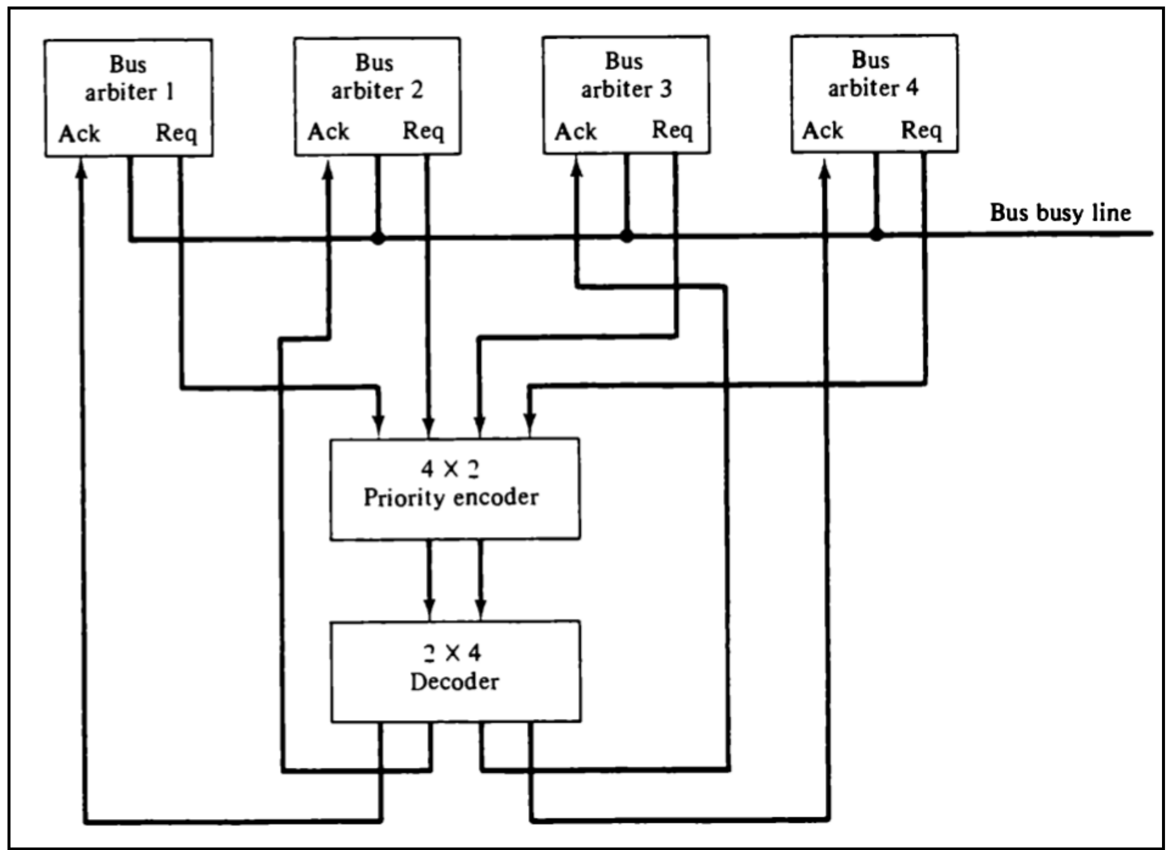


Figure 12.12 Parallel arbitration

Figure 12.12 shows the request lines from four arbiters going into a 4 x 2 priority encoder. The output of the encoder generates a 2-bit code which represents the highest-priority unit among those requesting the bus. The 2-bit code from the encoder output drives a 2 x 4 decoder which enables the proper acknowledge line to grant bus access to the highest-priority unit.

12.5.3 Dynamic Arbitration Algorithms

The two bus arbitration procedures just described use a static priority al-gorithm since the priority of each device is fixed by the way it is connected to the bus. In contrast, a dynamic priority algorithm gives the system the capability for changing the priority of the devices while the system is in operation. We now discuss a few arbitration procedures that use dynamic priority algorithms.

- **Time Slice Algorithm:** The time slice algorithm allocates a fixed-length time slice of bus time that is offered sequentially to each processor, in round-robin fashion. The service given to each system component with this scheme is independent of its loca-tion along the bus. No preference is given to any particular device since each is allotted the same amount of time to communicate with the bus.
- **Polling:** In a bus system that uses polling, the bus grant signal is replaced by a set of lines called poll lines which are connected to all units. These lines are used by the bus controller to define an address for each device connected to the bus. The bus controller sequences through the addresses in a prescribed manner. When a processor that requires access recognizes its address, it activates the bus busy line and then accesses the bus. After a number of bus cycles, the polling process continues by choosing a different processor. The polling sequence is normally programmable, and as a result, the selection priority can be altered under program control.
- **Least Recently Used (LRU):** The least recently used (LRU) algorithm gives the highest priority to the requesting device that has not used the bus for the longest interval. The priorities are adjusted after a number of bus cycles according to the LRU algorithm. With this procedure, no processor is favored over any other since the priorities are dynamically changed to give every device an opportunity to access the bus.
- **FIFO:** In the first-in, first-out (FIFO) scheme, requests are served in the order received. To implement this algorithm, the bus controller establishes a queue arranged according to the time that the bus requests arrive. Each processor must wait for its turn to use the bus on a first-in, first-out (FIFO) basis.
- **Rotating Daisy-Chain:** The rotating daisy-chain procedure is a dynamic extension of the daisy-chain algorithm. In this scheme there is no central bus controller, and the priority line is connected from the priority-out of the last device back to the priority-in of the first device in a closed loop.

This is similar to the connections shown in figure 11.11 except that the PO output of arbiter 4 is connected to the PI input of arbiter 1. Whichever device has access to the bus serves as a bus controller for the following arbitration. Each arbiter priority for a given bus cycle is determined by its position along the bus priority line from the arbiter whose processor is currently controlling the bus. Once an arbiter releases the bus, it has the lowest priority.

12.6 Inter-Processor Communication And Synchronization

A multiprocessor system has various processors that must be provided with a facility to communicate with each other. Using a common I/O channel, a communication path is established. The most frequently used procedure in a shared memory multiprocessor system is to set aside a part of the memory that is available to all processors. The major use of the common memory is to work as a message center similar to a mailbox, where every processor can leave messages for other processors and pick up messages meant for it.

The sending processor prepares a request, a message, or a procedure, and then places it in the memory mailbox. The receiving processor can check the mailbox periodically to determine if there are valid messages in it, as a processor identifies a request only while polling messages. However, the response time of this procedure may be time consuming. The sending processor has a more efficient procedure, and the procedure involves alerting the receiving processor directly using an interrupt signal. This procedure is achieved with the help of software-initiated interprocessor interrupt initialized in one processor, which when implemented generates an external interrupt condition in a second processor. This interrupt informs the second processor that processor one has inserted a new message in its mailbox.

A multiprocessor system has other shared resources in addition to shared memory. For example, An IOP to which a magnetic disk storage unit is connected, is available to all CPUs. This helps in providing a facility for sharing of system programs stored in the disk. A communication path can be

established between two CPUs through a link between two IOPs, which connects two different CPUs. This kind of link allows each CPU to treat the other as an I/O device, such that messages can be transferred through the I/O path.

There should be a provision for assigning resources to processors to avoid inconsistent use of shared resources by many processors. This job is given to the operating system. The three organizations that are used in the design of operating system for multiprocessors include:

- ***Master-slave configuration:*** In a master-slave configuration mode, one processor, designated the master, always implements the operating system functions. The remaining processors, designated as slaves, do not execute operating system functions. If a slave processor requires an operating system service, then it should request it by interrupting the master.
- ***Separate operating system:*** Each processor can implement the operating system routines that it requires in the separate operating system organization. This kind of organization is more appropriate for loosely-coupled systems wherein, every processor needs to have its own copy of the entire operating system.
- ***Distributed operating system:*** The operating system routines are shared among the available processors in the distributed operating system organization. However, each operating system function is allocated to only one processor at a time. This kind of organization is also termed as a floating operating system because the routines float from one processor to another, and the implementation of the routines are allocated to different processors at different times.

The memory is distributed among the processors and there is no shared memory for sending information in a loosely-coupled multiprocessor system. Message passing system through I/O channels is used for communication between processors. The communication is started by one processor calling a

procedure that exists in the memory of the processor with which it has to communicate. A communication channel is established when both the sending processor and the receiving processor recognize each other as source and destination. A message is then sent to the nodes with a header and different data objects that are required for communication between the nodes. In order to send the message between any two nodes, several possible paths are available. The operating system of each node has the routing information which indicates the available paths to send a message to different nodes. The communication efficiency of the interprocessor network depends on four major factors: Communication routing protocol, Processor speed, Data link speed, and Topology of the network.

Interprocessor Synchronization

Synchronization is a communication of control information between processors. Synchronization helps to:

- Implement the exact sequence of processes.
- Ensure mutually exclusive access to allocated writable data.

Synchronization refers to a special case where the control information is the data employed to communicate between processors. Synchronization is necessary to implement the exact sequence of processes and to ensure mutually exclusive access to shared writable data.

There are many mechanisms in multiprocessor systems to handle the synchronization of resources. The hardware directly implements low-level primitives. These primitives act as essential mechanisms that enforce mutual exclusion for more difficult mechanisms executed in software. Many hardware mechanisms for mutual exclusion are developed. However, the use of a binary semaphore is considered to be one of the most popular mechanisms.

Synchronization can be achieved by ***mutual exclusion with a semaphore***. Semaphores are considered to be the means of addressing the requirements of both task synchronization and mutual exclusion. Mutual exclusion includes a processor to eliminate or lock out access to allocated resources by other

processors when it is in a ***Critical Section***.

Mutual Exclusion with a Semaphore

Appropriately operating a multiprocessor system must provide a mechanism that would ensure systematic access to shared memory and other shared resources. This is required to protect data, since two or more processors can change the data simultaneously. This mechanism is referred to as ***mutual exclusion***. A multiprocessor system must have mutual exclusion to allow one processor to rule out or lock out access to an allocated resource by other processors when it is in a critical section. A critical section is defined as a program sequence which once started must complete implementation before another processor accesses the same allocated resource.

When the semaphore is set to one, it indicates that a processor is implementing a critical program, and the shared memory is unavailable to other processors. When the semaphore is set to zero, it indicates that the shared memory is available to any requesting processor. Processors sharing the same memory segment agree to not use the memory segment unless the semaphore is 0, showing that memory is available. The processors also concur to set the semaphore to 1, while they are implementing a critical section, and then to clear it to 0 when they are done.

Testing and setting the semaphore is considered to be a critical function, and needs to be carried out as a single indivisible operation. Otherwise, two or more processors may check the semaphore simultaneously and set the semaphore in such a way that it can enter a critical section at the same time. This action allows the simultaneous execution of these critical sections resulting in incorrect initialization of control factors and a loss of necessary information.

A semaphore is initialized using a test and set instruction together with a ***hardware lock*** mechanism. A hardware lock is defined as a processor-generated signal that helps in preventing other processors from using the system bus as long as the signal is active. When the instruction is being executed, the test-and-set instruction tests and sets a semaphore and activates

the lock mechanism. This helps in preventing the other processors from changing the semaphore between the time that the processor is testing it and the time that the processor is setting it.

Consider that the semaphore is a bit in the least significant position of a memory word whose address is symbolized by ***SEM***. Let the mnemonic ***TSL*** designate the "test and set while locked" function. The instruction

TSL SEM

is executed in two memory cycles, that is, the first one to read and the second to write without any interference as given below:

$R \leftarrow M[SEM]$ Test semaphore

$M[SEM] \leftarrow 1$ Set semaphore

In order to test the semaphore, its value is transferred to a processor register R and then set to 1. The value of R indicates what to do next. If the processor identifies that R = 1, it means that the semaphore was initially set. Even if the register is set again, it does not change the value of the semaphore. This indicates that another processor is executing a critical section and therefore, the processor that checked the semaphore does not access the shared memory. The common memory or the shared resource that the semaphore represents is available when R = 0. In order to avoid other processors from accessing memory, the semaphore is set to 1. Now, it is possible for the processor to execute the critical section. To release the shared resource to other processors, the final instruction of the program must clear location SEM to zero.

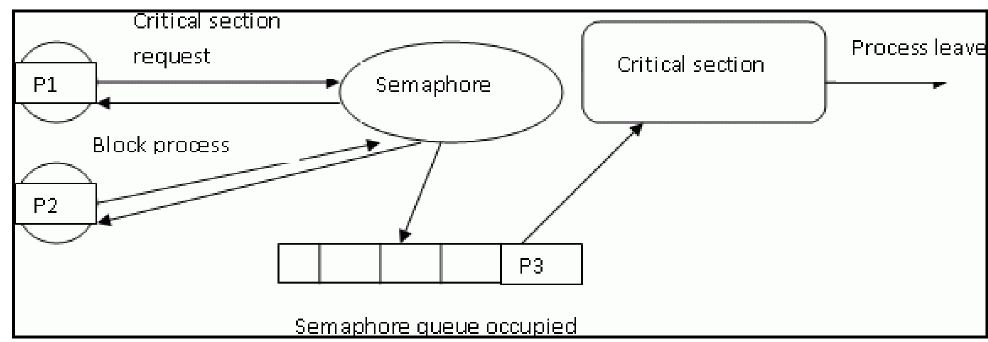


Figure 12.13 Mutual Exclusion with a Semaphore

It is crucial to note that the lock signal must be active at the time of execution of the test-and-set instruction. Once the semaphore is set, the lock signal does not have to be active. Therefore, the lock mechanism prevents other processors from accessing memory while the semaphore is being set. Once set, the semaphore itself will prevent other processors from accessing shared memory while one processor is implementing a critical section.

12.7 Symmetric Multiprocessors

Virtually all single-user personal computers and most workstations contained a single general-purpose microprocessor. As demands for performance increase and as the cost of microprocessors continues to drop, vendors have introduced systems with an ***Symmetric multiprocessor (SMP)*** organization. The term SMP refers to a computer hardware architecture and also to the operating system behavior that reflects that architecture.

An SMP can be defined as a standalone computer system with the following characteristics:

- There are two or more similar processors of comparable capability.
- These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme, such that memory access time is approximately the same for each processor.
- All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.
- All processors can perform the same functions (hence the term symmetric).
- The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.

The operating system of an SMP schedules processes or threads across all of the processors. An SMP organization has a number of potential advantages over a uniprocessor organization, including the following:

- ***Performance:*** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type.
- ***Availability:*** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.
- ***Incremental growth:*** A user can enhance the performance of a system by adding an additional processor.
- ***Scaling:*** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

It is important to note that these are potential, rather than guaranteed, benefits. The operating system must provide tools and functions to exploit the parallelism in an SMP system.

An attractive feature of an SMP is that the existence of multiple processors is transparent to the user. The operating system takes care of scheduling of threads or processes on individual processors and of synchronization among processors.

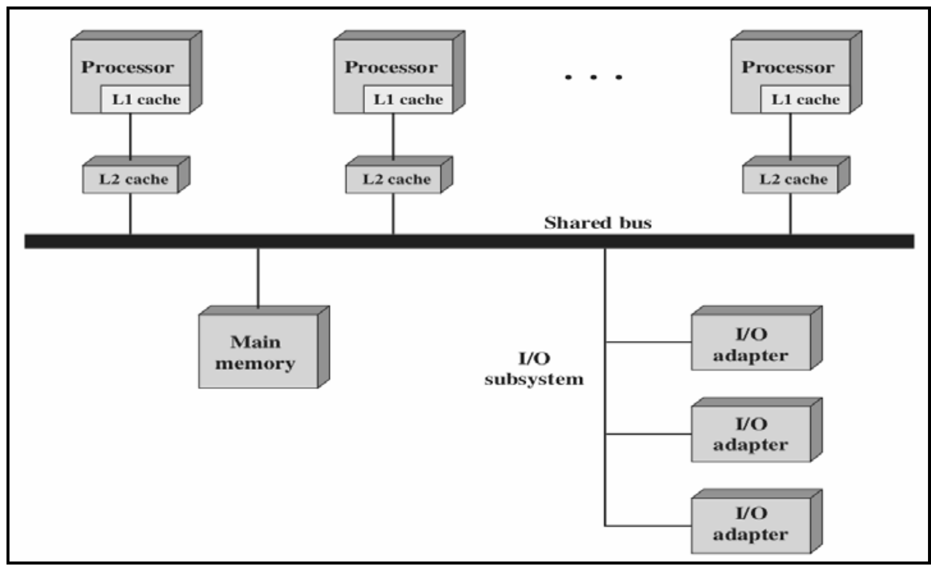


Figure 12.14 Symmetric Multiprocessor Organization

12.8 Summary

- A multiprocessor generally refers to a single computer that has many processors. The term 'multiprocessor' can also be used to describe several separate computers running together. It is also referred to as clustering.
- The difference that exists between multicomputer systems and multiprocessors depends on the extent of resource sharing and cooperation in solving a problem.
- There are two types of multiprocessor systems; they are tightly-coupled multiprocessor systems and loosely-coupled multiprocessor systems.
- Multiprocessor systems work efficiently as high-performance database servers, Internet servers, and network servers.
- A multiport memory system uses separate buses between each CPU and each memory module.
- The term symmetric multiprocessor refers to a computer hardware architecture and also to the operating system behavior that reflects that architecture.
- Semaphores are considered to be the means of addressing the

requirements of both task synchronization and mutual exclusion.

- A bus that connects major components in a multi-processor system, such as CPUs, lOPs, and memory, is called a system bus.

12.9 Key Terms

- ***Autonomous Computers:*** A network administered by a single set of management rules that are controlled by a single person, group, or organization. Autonomous systems frequently use only one routing protocol even though it is possible to use multiple protocols.
- ***Control Logic:*** It is the part of a software architecture that helps in controlling what the program will do. This part of the program is also termed as controller.
- ***Multithreading:*** It is a process wherein the same job is broken logically and performed simultaneously and the output is combined at the end of processing.
- ***Real-time Applications:*** A real-time application is an application program that works within a given time frame that the user assumes as immediate or current.

12.10 Check Your Progress

Q1) What are the major characteristics of a multiprocessor? Also list its advantages.

Q2) Differentiate between tightly-coupled and loosely-coupled multiprocessor systems.

Q3) Write a short note on Hypercube Interconnection.

Q4) What is a dynamic arbitration algorithm? Discuss any two of them in detail.

Q5) What are symmetric multiprocessors? How is it beneficial than a uniprocessor organization?

References:

Computer Organization and Architecture, *9th edition*, William Stallings, Pearson Publication.

Computer System Architecture, M. Morris Mano